

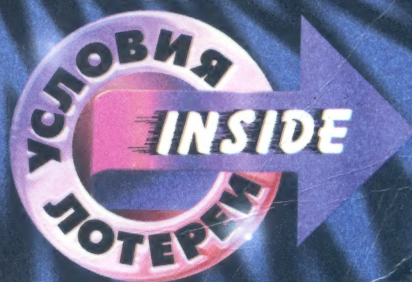
Фаронов В.В.

TurboPascal 7.0

Н А Ч А Л Ь Н Ы Й К У Р С
учебное пособие



Нолидж
KNOWLEDGE
W



Наши подарки и Лотерея

Уважаемые читатели! У Вас есть возможность бесплатно получить одну из книг, внесенную в соответствующий список, либо приобрести со скидкой 10% любую из книг, предлагаемых книготорговым объединением «НОЛИДЖ-РИДАС».

Это можно сделать, заполнив специальную форму (анкету) на нашем сайте **WWW.KNOWLEDGE.RU** или аналогичный бланк, прилагаемый к нашим новым книгам. К рассмотрению будут приниматься только анкеты, содержащие корректную информацию в обязательных полях. Для получения книги Вам необходимо приехать с заполненным бланком (или его ксерокопией) в фирму «Ридас». Если Вы заполнили форму на сайте, то при получении книги необходимо предоставить распечатку результатов ввода данных (они выводятся на экран после нажатия кнопки «Внести»).

Список книг, предлагаемых бесплатно, можно посмотреть на нашем сайте. Приобретая книгу со скидкой в 10%, Вы делаете покупку, которая в результате оказывается на 40-50% дешевле, чем в любом магазине, учитывая оптовые цены нашего книготоргового объединения. Приславшие анкету автоматически становятся участниками нашей лотереи.

Лотерея

Итак, Вы стали счастливым обладателем книги, выпущенной издательством «Нолидж». Теперь вы можете не только получить в подарок книгу и стать участником розыгрыша новых книг, но и выиграть в нашей лотерее ценный приз – современный персональный компьютер, лазерный принтер, устройство DVD и т.д..

Для этого вам надо отрезать нижний красный уголок с задней стороны обложки этой книги и вместе с заполненной анкетой прислать по адресу: 113114, Москва, Шлюзовая набережная, 10, издательство «Нолидж», или привезти анкету с уголком в фирму «Ридас».

ВНИМАНИЕ! На передней стороне обложки всех книг с уголком изображена большая красная стрелка – это поможет Вам сразу узнать книги нашего издательства на полке в магазине.

Каждый человек, приславший нам красный уголок книги, заносится в базу данных для участия в розыгрыше призов. Если Вы укажете адрес вашей электронной почты, то обязательно получите письмо, подтверждающее Ваше участие в лотерее.

ВНИМАНИЕ! Каждый присланный Вами уголок участвует в розыгрыше призов отдельно, повышая Ваши шансы на выигрыш.

Первый розыгрыш призов состоится 13 апреля 1999 года, в день 3-летия издательства «Нолидж».

Издательство оставляет за собой право не принимать к рассмотрению анкеты с незаполненными обязательными полями, а так же оформленными неразборчиво или несерьезно! Ответы на вопросы анкеты очень важны для нас при выработке издательской политики, с тем, чтобы издавалось больше интересных и нужных Вам книг.

Издательство уведомляет, что фамилии и фотографии людей, выигравших призы в нашей лотерее, могут быть использованы нами в рекламных целях в книгах, журналах или в Internet.

Лица, приславшие 4 и более красных уголков, становятся участниками розыгрыша лотереи не только текущего года, но и всех последующих. Кроме того, учреждается специальный ежегодный приз для самого верного читателя «Нолидж».

У Вас есть возможность получать свежий прайс-лист по электронной или обычной почте. Вы можете стать заказчиком нашей системы «книга-почтой» и оперативно получать любые интересующие Вас книги нашего и других издательств по почте в любом уголке мира. Отправьте соответствующее письмо.

Заранее Вас благодарим!

В.В.Фаронов

Турбо Паскаль
Начальный курс
Учебное пособие

Издательство «Нолидж» 1999

ББК 32.973 я7

Ф.24

УДК 681.3.06 (095)

Фаронов В.В.

Ф24 Турбо Паскаль 7.0. Начальный курс. Учебное пособие.-М.: «Нолидж», 1999. - 616 с., ил.

В книге содержится исчерпывающее руководство по программированию в среде Турбо Паскаль 7.0 корпорации Borland. Изложение ведется от простого к сложному, позволяя изучить систему программирования начинающим и пополнить свои знания опытным программистам.

ISBN 5-89251-054-9

Лицензия ЛР № 064480 от 04 марта 1996 г.

© Фаронов В.В., 1998.

© Нолидж, 1999.

Оглавление

От автора	9
Часть 1. Ядро Турбо Паскаля	11
Глава 1. Знакомство со средой Турбо Паскаля	13
1.1. Как начать работу с Турбо Паскалем	13
1.2. Функциональные клавиши	15
1.3. Текстовый редактор	16
1.4. Основные приемы работы в среде Турбо Паскаля	18
1.4.1. Работа с файлами	18
1.4.2. Прогон и отладка программы	19
1.4.3. Справочная служба Турбо Паскаля	20
Глава 2. Знакомство с языком Турбо Паскаль	22
2.1. Ваша первая программа	22
2.2. Типы данных	27
2.3. Преобразования типов и действия над ними	31
2.4. Операторы языка	36
2.4.1. Составной оператор и пустой оператор	36
2.4.2. Условный оператор	37
2.4.3. Операторы повторов	39
2.4.4. Оператор выбора	42
2.4.5. Метки и операторы перехода	44
2.5. Массивы	45
2.6. Процедуры и функции	47
2.7. Примеры программ	53
2.7.1. Вычисление дня недели	54
2.7.2. Биоритмы	58
2.7.3. Игра ним	66
Глава 3. Элементы языка	81
3.1. Алфавит	81
3.2. Идентификаторы	82
3.3. Константы	82
3.4. Выражения	84
3.5. Операции	85
3.6. Структура программы	88
Глава 4. Типы данных	90
4.1. Простые типы	91
4.1.1. Порядковые типы	91
4.1.2. Вещественные типы	99
4.2. Структурированные типы	102
4.2.1. Массивы	102
4.2.2. Записи	104
4.2.3. Множества	108
4.3. Строки	112
4.4. Совместимость и преобразование типов	115
Глава 5. Файлы	119
5.1. Доступ к файлам	120
5.1.1. Имена файлов	120
5.1.2. Логические устройства	121
5.1.3. Инициация файла	123
5.2. Процедуры и функции для работы с файлами	124
5.3. Текстовые файлы	131
5.4. Типизированные файлы	136
5.5. Нетипизированные файлы	138

Глава 6. Указатели и динамическая память	140
6.1. Динамическая память	140
6.2. Адреса и указатели	140
6.3. Объявление указателей	141
6.4. Выделение и освобождение динамической памяти	143
6.5. Использование указателей	148
6.6. Процедуры и функции для работы с динамической памятью	151
6.7. Администратор кучи	154
Глава 7. Типизированные константы	157
7.1. Константы простых типов и типа <i>String</i>	157
7.2. Константы-массивы	158
7.3. Константы-записи	159
7.4. Константы-множества	160
7.5. Константы-указатели	160
Глава 8. Процедуры и функции	161
8.1. Локализация имен	161
8.2. Описание подпрограммы	165
8.2.1. Заголовок	165
8.2.2. Параметры	167
8.3. Параметры-массивы и параметры-строки	171
8.4. Процедурные типы. Параметры-функции и параметры-процедуры	173
8.5. Нетипизированные параметры-переменные	176
8.6. Рекурсия и опережающее описание	178
8.7. Расширенный синтаксис вызова функций	180
Глава 9. Модули	182
9.1. Структура модулей	183
9.2. Заголовок модуля и связь модулей друг с другом	183
9.3. Интерфейсная часть	184
9.4. Исполняемая часть	184
9.5. Иницилирующая часть	185
9.6. Компиляция модулей	186
9.7. Доступ к объявленным в модуле объектам	187
9.8. Стандартные модули	189
Глава 10. Объекты	191
10.1. Основные принципы ООП	192
10.2. Постановка учебной задачи	193
10.3. Создание объектов	194
10.4. Использование объектов	202
Глава 11. Другие возможности Турбо Паскаля	208
11.1. Внешние процедуры (функции)	208
11.2. Использование встроенных машинных кодов	209
11.3. Обращение к функциям операционной системы	212
11.4. Поддержка процедур обработки прерываний	214
11.5. Запуск внешних программ	216
11.6. Оверлей	219
11.7. Прямое обращение к памяти и портам ввода-вывода	224
11.8. Длинные строки	225
Глава 12. Встроенный ассемблер	233
12.1. Общее описание МП 8086/8088	233
12.1.1. Регистры	234
12.1.2. Адресация	236
12.1.3. Система команд	238
12.2. Специфика встроенного ассемблера	247
12.2.1. Оператор <i>ASM</i>	247
12.2.2. Синтаксис ассемблерных команд	249
12.2.3. Директивы ассемблера	258

12.2.4. Ассемблерные подпрограммы	260
Глава 13. Использование библиотеки CRT	262
13.1. Программирование клавиатуры	262
13.2. Текстовый вывод на экран	265
13.3. Программирование звукового генератора	274
Глава 14. Использование библиотеки Graph	276
14.1. Переход в графический режим и возврат в текстовый	276
14.1.1. Краткая характеристика графических режимов работы дисплейных адаптеров	276
14.1.2. Процедуры и функции	278
14.2. Координаты, окна, страницы	284
13.3. Линии и точки	291
14.4. Многоугольники	299
14.5. Дуги, окружности, эллипсы	301
14.6. Краски, палитры, заполнения	307
14.7. Сохранение и выдача изображений	323
14.8. Вывод текста	326
14.9. Включение драйвера и шрифтов в тело программы	334
Часть 2. Библиотека Turbo Vision	337
Глава 15. Введение в Turbo Vision	339
15.1. Простейшая программа в Turbo Vision	339
15.2. Формирование строки статуса	341
15.3. Формирование меню	344
15.4. Команды	347
15.5. События и их обработка	348
15.6. Программирование диалоговых запросов	352
15.7. Инкапсуляция новых полей и методов	355
15.8. Создание и использование групп	358
15.9. Вывод текста	361
15.10. Цветовая палитра	364
15.11. Использование коллекций	367
15.12. Указатель на элемент списка	371
15.13. Диалоговое окно выбора режима	374
15.14. Обработка команд пользователя	380
15.15. Редактирование и добавление записей	381
15.16. Удаление записи	385
15.17. Режим поиска записи	385
15.18. Итоги	387
Глава 16. Общая характеристика объектов	389
16.1. Структура объектов	390
16.2. Абстрактные объекты и методы	390
16.3. Функциональность объектов	391
16.4. Обзор видимых элементов	393
16.4.1. Группы видимых элементов	393
16.4.2. Терминальные видимые объекты	395
16.5. Невидимые элементы	399
16.5.1. Потоки	399
16.5.2. Коллекции	400
16.5.3. Списки строк	401
16.5.4. Контролеры	401
Глава 17. Видимые элементы	403
17.1. Территориальность	403
17.2. Вывод изображения	405
17.2.1. Заполнение области	405
17.2.2. Цвет изображения	406
17.3. Группы	410
17.3.1. Создание группы и изменение ее состава	411

17.3.2. Z-упорядочение и дерево видимых элементов	412
17.3.3. Активные элементы	414
17.4. Модальные видимые элементы	415
17.5. Изменение свойств элемента	416
17.5.1. Поле Options	416
17.5.2. Поле GrowMode	419
17.5.3. Поле DragMode	420
17.5.4. Поле State	421
17.5.5. Воздействие на состояние поля State	423
Глава 18. События	425
18.1. Природа событий	425
18.2. Виды событий	426
18.2.1. События от мыши	426
18.2.2. События от клавиатуры	427
18.2.3. Сообщения	427
18.2.4. Пустые события	428
18.3. Маршрутизация событий	428
18.3.1. Позиционированные события	429
18.3.2. Общие события	430
18.4. Фаза событий	430
18.5. Команды	432
18.5.1. Преобразование активных событий в команды	432
18.5.2. Запрещение и разрешение команд	435
18.6. Модификация и обработка событий	435
18.6.1. События, определенные пользователем	435
18.6.2. Маскирование и очистка событий	436
18.6.3. Перекрытие HandleEvent	437
18.6.4. Перекрытие GetEvent	438
18.6.5. Неиспользованное время	438
18.6.6. Ненужные события	440
18.7. Взаимодействие видимых элементов	440
18.8. Контекстная помощь	442
Глава 19. Коллекции	453
19.1. Элементы коллекций	453
19.2. Создание коллекций	454
19.3. Доступ к элементам коллекций	456
19.4. Отсортированные коллекции	460
19.5. Коллекции строк	464
19.6. Полиморфные коллекции	467
19.7. Коллекции и управление памятью	471
Глава 20. Потоки	473
20.1. Сущность потоков	473
20.2. Регистрация объектов	474
20.3. Создание и удаление потока	476
20.4. Работа с потоком	478
20.4.1. Методы Put и Get	479
20.4.2. Методы Store и Load	479
20.4.3. Обработка ошибок	481
20.4.4. Прямой доступ к потокам	481
20.4.5. Использование потоков с произвольными данными	482
Глава 21. Ресурсы Turbo Vision	483
21.1. Создание ресурса	483
21.2. Использование ресурса	484
21.3. Строковые ресурсы	485
Глава 22. Объекты-контролеры	488
22.1. Типы объектов-контролеров	488
22.1. . TPXPictureValidator	488

22.1.2. TFilterValidator.....	489
22.1.3. TRangeValidator.....	489
22.1.4. TLookupValidator.....	489
22.1.5. TStringLookupValidator.....	490
22.2. Использование объектов-контролеров.....	490
22.2.1. Контроль текстового ввода.....	490
22.2.2. Проверка других объектов.....	491
22.2.3. Реализация нестандартного контроля.....	492
Глава 23. Практика использования.....	493
23.1. Контроль за динамической памятью.....	493
23.2. Обработка ошибок инициации и модальных состояний.....	495
23.3. Отладка программ.....	496
23.4. Использование оверлея.....	500
23.5. Порядок вызова наследуемого метода.....	502
23.5.1. Конструктор.....	503
23.5.2. Деструктор.....	504
23.5.3. Другие методы.....	505
23.6. Примеры программных реализаций.....	506
23.6.1. Строка статуса.....	506
23.6.2. Меню.....	508
23.6.3. Диалоговое окно.....	510
23.6.4. Окно с текстом.....	514
23.6.5. Окно со скроллером.....	518
23.6.6. Просмотр списка файлов.....	521
Приложения.....	525
П1. Среда Турбо Паскаля.....	525
П1.1. Элементы диалоговой среды.....	525
П1.1.1. Работа с окнами.....	526
П1.1.2. Работа с меню.....	526
П1.1.3. Работа с диалоговым окном.....	527
П1.2. Система меню.....	528
П1.2.1. Меню опции FILE.....	528
П1.2.3. Меню опции EDIT.....	531
П1.2.4. Меню опции SEARCH.....	531
П1.2.5. Меню опции RUN.....	533
П1.2.6. Меню опции COMPILE.....	534
П1.2.7. Меню опции DEBUG.....	535
П1.2.8. Меню опции TOOLS.....	538
П1.2.9. Меню опции OPTIONS.....	538
П1.2.9. Меню опции WINDOW.....	546
П1.2.10. Меню опции HELP.....	546
П1.3. Директивы компилятора.....	547
П1.4. Редактор.....	549
П1.4.1. Команды перемещения курсора.....	549
П1.4.2. Команды удаления/вставки.....	550
П1.4.3. Команды работы с блоками.....	550
П1.4.4. Прочие команды.....	550
П1.4.5. Команды, передаваемые среде из редактора.....	551
П1.5. Вызов Турбо-Паскаля.....	552
П2. Варианты кодировки знакогенераторов ПК.....	553
П3. Сообщения и коды ошибок.....	555
П3.1. Сообщения об ошибках периода компиляции.....	555
П3.2. Ошибки, возникающие во время выполнения программ.....	566
П3.2.1. Ошибки, обнаруживаемые ДОС.....	566
П3.2.2. Ошибки ввода-вывода.....	567
П3.2.3. Критические ошибки.....	568
П3.2.4. Фатальные ошибки.....	568

П4. Стандартные библиотечные модули	570
П4.1. Модуль System	570
П4.1.1. Константы	570
П4.1.2. Переменные	571
П4.1.3. Стандартные процедуры и функции	571
П4.2. Модуль DOS	576
П4.2.1. Константы	576
П4.2.2. Типы	576
П4.2.3. Переменные	577
П4.2.4. Процедуры и функции	578
П4.3. Модуль CRT	580
П4.3.1. Константы	580
П4.3.2. Переменные	580
П4.3.3. Процедуры и функции	581
П4.4. Модуль Graph	582
П4.4.1. Константы	583
П4.4.2. Типы	586
П4.4.3. Переменные	586
П4.4.4. Процедуры	586
П4.4.5. Функции	589
П5. Тексты программ	591
П5.1. Программа определения дня недели	591
П5.2. Определение биоритмов	591
П5.3. Игра ним	595
П5.4. Программа Notebook	600
Литература	611

От автора

Система программирования Турбо Паскаль, разработанная американской корпорацией *Borland*, остается одной из самых популярных систем программирования в мире. Этому способствуют, с одной стороны, простота лежащего в ее основе языка программирования Паскаль, а с другой — труд и талант сотрудников *Borland* во главе с идеологом и создателем Турбо Паскаля Андерсом Хейлсбергом, приложивших немало усилий к ее совершенствованию. Придуманый швейцарским ученым Никласом Виртом как средство для обучения студентов программированию, язык Паскаль стараниями А.Хейлсберга превратился в мощную современную профессиональную систему программирования, которой по плечу любые задачи — от создания простых программ, предназначенных для решения несложных вычислительных задач, до разработки сложнейших реляционных систем управления базами данных. Когда в далеком 1986 году я начинал работу с Турбо Паскалем, я не мог предположить, какую огромную роль в моей жизни сыграет эта замечательная система программирования. Сегодня Турбо Паскаль — мой самый любимый профессиональный инструмент. Появление *Windows* и инструментальных средств *Borland Pascal with Objects* и *Delphi* для разработки программ в среде *Windows* лишний раз показало, какие поистине неисчерпаемые возможности таит он в себе: и *Borland Pascal*, и используемый в *Delphi* язык *Object Pascal* основываются на Турбо Паскале и развивают его идеи.

В первой части этой книги описывается ядро системы программирования — язык Турбо Паскаль и стандартные библиотеки *CRT* и *GRAPH*. Эта часть рассчитана, в основном, на начинающих программистов желающих овладеть азами искусства программирования и попробовать свои силы в решении относительно несложных задач.

Во второй части рассматривается объектно-ориентированная библиотека *Turbo Vision*, предназначенная для разработки современных диалоговых программ средней сложности.

Проблемы расширения стандартных средств Турбо Паскаля и разработки собственного инструментария для гибкого управления техническими возможностями персонального компьютера, а также практические рекомендации по программированию в защищенном режиме работы процессора и адаптации Ваших программ к работе в среде *Windows* рассматриваются в другой моей книге — «Турбо Паскаль. Расширение возможностей», которая дополняет эту и, как и она, выпускается издательством «Нолидж».

В основу книги положен материал, опубликованный мною ранее в других книгах, посвященных Турбо Паскалю. При подготовке этого издания он был дополнен и исправлен, однако большая его часть осталась без изменения, т.к. остался без изменения и сам Турбо Паскаль. Впереди — *Delphi*, но это тема другой книги.

4 ноября 1996 г.

ЧАСТЬ 1

ЯДРО ТУРБО ПАСКАЛЯ

Глава 1

ЗНАКОМСТВО СО СРЕДОЙ ТУРБО ПАСКАЛЯ

Система программирования Турбо Паскаль представляет собой единство двух в известной степени самостоятельных начал: компилятора с языка программирования Паскаль (язык назван в честь выдающегося французского математика и философа Блеза Паскаля (1623 — 1662)) и некоторой инструментальной программной оболочки, способствующей повышению эффективности создания программ. Для краткости условимся в дальнейшем называть реализуемый компилятором язык программирования Паскаль языком *Турбо Паскаль*, а разнообразные сервисные услуги, предоставляемые программной оболочкой — *средой Турбо Паскаля*.

Среда Турбо Паскаля — это первое, с чем сталкивается любой программист, приступающий к практической работе с системой. (Если по каким-либо причинам Вы не собираетесь писать собственные программы, можно пропустить эту главу, в которой приводятся минимальные сведения об основных приемах работы в среде Турбо Паскаля. Более полные сведения о ней содержатся в прил.1.)

1.1. КАК НАЧАТЬ РАБОТУ С ТУРБО ПАСКАЛЕМ

Система Турбо Паскаль довольно значительна по объему. Она поставляется на нескольких дистрибутивных дискетах и устанавливается на жесткий диск. При развертывании системы на жестком диске обычно создается отдельный каталог с именем *TP* (или *PAS*, *TURBOPAS*, *PASCAL* и т.п.), в который помещаются все файлы с дистрибутивных дискет. Для вызова Турбо Паскаля необходимо отыскать в древовидной структуре каталогов ПК этот каталог и в нем файл с именем *TURBO.EXE*. Этот файл содержит готовую к работе диалоговую систему программирования Турбо Паскаль. В него входят минимально необходимые части Турбо Паскаля (текстовый редактор, компилятор, компоновщик, загрузчик). Для нормальной работы в диалоговой среде понадобятся также основная библиотека, располагающаяся в файле *TURBO.TPL*, и справочная служба (файл *TURBO.HLP*). В принципе, этих файлов достаточно для написания, компиляции и исполнения большинства примеров, содержащихся в этой книге.

Пусть перечисленные файлы располагаются в каталоге *TP* на диске *D*. Тогда для вызова Турбо Паскаля следует дать команду

D:\TP\TURBO

По этой команде операционная система *MS-DOS* поставит на исполнение программу из файла *TURBO.EXE*: загрузит программу в оперативную память и передаст ей управление.

Не рекомендуется работать с системой, назначив в качестве каталога по умолчанию (текущего каталога) тот, в котором хранятся перечисленные выше файлы (этот каталог будем называть *системным*). Во-первых, в

таком случае можно ошибочно стереть какой-либо из файлов системы программирования и тем самым нарушить ее работоспособность, а во-вторых, этот каталог очень скоро заполнится другими файлами, прямо не относящимися к Турбо Паскалю. Существует и еще одна причина, по которой нежелательно работать в системном каталоге. Дело в том, что Турбо Паскаль имеет свойство «запоминать» свою настройку в двух файлах с именами *TURBO.TP* и *TURBO.PCK*. При вызове система начинает поиск этих файлов в текущем каталоге. Если этот каталог — Ваш индивидуальный, система всякий раз будет настраиваться так, как Вы этого хотите. Если эти файлы не обнаружены в Вашем каталоге (а при первом обращении к Турбо Паскалю так оно и будет), система продолжит поиск в системном каталоге, а не найдя их там, настроится стандартным образом. Впоследствии можно сохранить настроечные файлы в своем каталоге и тем самым избавить себя от необходимости перенастройки системы всякий раз при обращении к ней.

После успешного вызова системы экран ПК приобретает вид, показанный на рис.1.1.



Рис.1.1. Вид экрана после вызова Турбо Паскаля

Сразу же скажем, что для выхода из Турбо Паскаля следует нажать клавишу *Alt* и, не отпуская ее, — клавишу с латинской буквой *X*, после чего можно отпустить обе клавиши.

Верхняя строка содержит «меню» возможных режимов работы Турбо Паскаля, нижняя — краткую справку о назначении основных функций — клавиш. Вся остальная часть экрана принадлежит окну редактора, очерченному двойной рамкой и предназначенному для ввода и коррекции текста программ. В его верхней строке приводятся имя того дискового файла, откуда был прочитан текст программы (новому файлу присваивается имя *NONAME00.PAS*), два специальных поля, используемых при работе с устройством ввода «мышь» (эти поля выделены квадратными

скобками), и цифра 1 — номер окна. В Турбо Паскале можно работать одновременно с несколькими программами (или частями одной крупной программы), каждая из которых может располагаться в отдельном окне редактора. Среда позволяет использовать до девяти окон редактора одновременно.

Кроме окна (окон) редактора, в Турбо Паскале используются также окна: отладочного режима, вывода результатов работы программы, справочной службы, стека, регистров. По желанию они могут вызываться на экран поочередно или присутствовать на нем одновременно.

1.2. ФУНКЦИОНАЛЬНЫЕ КЛАВИШИ

Функциональные клавиши используются для управления средой Турбо Паскаля. Они обозначаются *F1*, *F2*, ..., *F12* и располагаются в самом верхнем ряду клавиатуры. С каждой из этих клавиш связывается некоторая команда меню. Действие почти всех функциональных клавиш можно модифицировать тремя особыми клавишами: *Alt* (от *ALternative* — *гoлoвнoй*), *Ctrl* (*CoNTRoL* — *управление*) и *Shift* (*SHIFT* — *cгvур*). Эти клавиши используются подобно клавише временной смены регистра на пишущей машинке: нужно нажать на одну из них и затем, не отпуская ее, нажать функциональную клавишу. В дальнейшем такое совместное нажатие двух клавиш будем обозначать чертой. Например, *Alt-F3* означает, что вместе с клавишей *Alt* необходимо нажать клавишу *F3*, *Ctrl-F9* — вместе с *Ctrl* нажимается клавиша *F9* и т.д.

Ниже приводятся команды, которые передаются среде Турбо Паскаля функциональными клавишами и некоторыми их комбинациями с клавишами *Ctrl* и *Alt*:

F1 — обратиться за справкой к встроенной справочной службе (*Help* — *помощь*);

F2 — записать редактируемый текст в дисковый файл;

F3 — прочитать текст из дискового файла в окно редактора;

F4 — используется в отладочном режиме: начать или продолжить исполнение программы и остановиться перед исполнением той ее строки, на которой стоит курсор;

F5 — распахнуть активное окно на весь экран;

F6 — сделать активным следующее окно;

F7 — используется в отладочном режиме: выполнить следующую строку программы; если в строке есть обращение к процедуре (функции), войти в эту процедуру и остановиться перед исполнением первого ее оператора;

F8 — используется в отладочном режиме: выполнить следующую строку программы; если в строке есть обращение к процедуре (функции), исполнить ее и не проследивать ее работу;

F9 — компилировать программу, но не выполнять ее;

F10 — перейти к диалоговому выбору режима работы с помощью главного меню;

Ctrl-F9 — выполнить прогон программы: компилировать программу, находящуюся в редакторе, загрузить ее в оперативную память и выполнить, после чего вернуться в среду Турбо Паскаля;

Alt-F5 — сменить окно редактора на окно вывода результатов работы (прогона) программы.

Полное описание функциональных клавиш содержится в прил.1, а сейчас — самый краткий комментарий.

Во-первых, Вам понадобятся команды **Ctrl-F9** для проверки работы Вашей программы и **Alt-X** для выхода из Турбо Паскаля. Клавиши **F2** и **F3** помогут Вам в работе с Вашим каталогом. Командой **ALT-F5** Вы в любой момент сможете просмотреть данные, выданные на экран в результате прогона программы.

1.3. ТЕКСТОВЫЙ РЕДАКТОР

Текстовый редактор среды Турбо Паскаля предоставляет пользователю удобные средства создания и редактирования текстов программ. Признаком того, что среда находится в состоянии редактирования, является наличие в окне редактора курсора — небольшого мигающего прямо-угольника. Режим редактирования автоматически устанавливается сразу после загрузки Турбо Паскаля. Из режима редактирования можно перейти к любому другому режиму работы Турбо Паскаля с помощью функциональных клавиш или выбора нужного режима из главного меню. Если среда находится в состоянии выбора из меню, курсор исчезает, а в строке меню появляется цветной указатель — прямоугольник, выделяющий одно из кодовых слов. Для перехода от состояния выбора режима из главного меню в состояние редактирования нужно нажать клавишу **Esc** (*ESCAPE* — *ускользнуть, убежать*), а для перехода к выбору из главного меню — **F10**.

Рассмотрим основные приемы работы с текстовым редактором.

Для создания текста программы нужно ввести этот текст с помощью клавиатуры ПК подобно тому, как это делается при печатании текста на пишущей машинке. После заполнения очередной строки следует нажать на клавишу **Enter**, чтобы перевести курсор на следующую строку (курсор всегда показывает то место на экране, куда будет помещен очередной вводимый символ программы).

Окно редактора имитирует длинный и достаточно широкий лист бумаги, фрагмент которого виден в окне. Если курсор достиг нижнего края, осуществляется прокрутка окна редактора: его содержимое смещается вверх на одну строку и снизу появляется новая строка листа. Если курсор достиг правой границы экрана, окно начинает по мере ввода символов смещаться вправо, показывая правый край листа. Размеры листа по горизонтали и вертикали ограничиваются только общим числом символов в файле, которых не должно быть больше 64535, однако компилятор Турбо Паскаля воспринимает строки программы длиной не более 126 символов.

Окно можно смещать относительно листа с помощью следующих клавиш:

PgUp — на страницу вверх (*PaGe UP* — *страницу вверх*);

PgDn — на страницу вниз (*PaGe Down* — *страницу вниз*);

Home — в начало текущей строки (*HOME* — *домой*);

End — в конец текущей строки (*END* — *конец*);

Ctrl-PgUp — в начало текста;

Ctrl-PgDn — в конец текста.

Клавишами перевода курсора (эти клавиши помечены соответствующими стрелками и располагаются в правой части клавиатуры) его можно смещать по экрану. При достижении курсором границ окна оно смещается на строку или на символ.

Если Вы ошиблись при вводе очередного символа, его можно стереть с помощью клавиши, обозначенной стрелкой влево (клавиша *Backspace* располагается над клавишей *Enter*). Клавиша *Del* (от *DElete* — *стирать*) стирает символ, на который в данный момент указывает курсор, а команда *Ctrl-Y* — всю строку, на которой располагается курсор.

Следует помнить, что редактор Турбо Паскаля вставляет в конце каждой строки невидимый на экране символ-разделитель. Этот символ вставляется клавишей *Enter*, а стирается клавишами *Backspace* или *Del*. С помощью вставки/стирания разделителя можно «разрезать»/«склеить» строки. Чтобы «разрезать» строку, следует подвести курсор к нужному месту и нажать клавишу *Enter*, чтобы «склеить» соседние строки, нужно установить курсор в конец первой строки (для этого удобно использовать клавишу *End*) и нажать клавишу *Del* или установить курсор в начало следующей строки (клавишей *Home*) и нажать клавишу *Backspace*.

Нормальный режим работы редактора — режим вставки, в котором каждый вновь вводимый символ как бы «раздвигает» текст на экране, смещая вправо остаток строки. Следует учитывать, что «разрезание» и последующая вставка пропущенных строк возможны только в этом режиме. Редактор может также работать в режиме наложения новых символов на существующий старый текст: в этом режиме новый символ заменяет собой тот символ, на который указывает курсор, а остаток строки справа от курсора не смещается вправо. Для перехода к режиму наложения нужно нажать клавишу *Ins* (*INSert* — *вставка*), если нажать эту клавишу еще раз, вновь восстановится режим вставки. Признаком того, в каком режиме работает редактор, является форма курсора: в режиме вставки курсор похож на мигающий символ подчеркивания, а в режиме наложения он представляет собой крупный мигающий прямоугольник, заслоняющий символ целиком.

И еще об одной возможности редактора. Обычно редактор работает в режиме автоотступа. В этом режиме каждая новая строка начинается в той же позиции на экране, что и предыдущая. Режим автоотступа поддерживает хороший стиль оформления текстов программ: отступы от левого края выделяют тело условного или составного оператора и делают программу более наглядной. Отказаться от автоотступа можно командой *Ctrl-O I* (при нажатой клавише *Ctrl* нажимается сначала клавиша *O*, затем *O* отпускается и нажимается клавиша *I*), повторная команда *Ctrl-O I* восстановит режим автоотступа.

Ниже перечислены наиболее часто используемые команды текстового редактора Турбо Паскаля.

Смещение курсора

PgUp — на страницу вверх;

PgDn — на страницу вниз;

Home — в начало строки;

End — в конец строки;
Ctrl-PgUp — в начало текста;
Ctrl-PgDn — в конец текста.

Команды редактирования

Backspace — стереть символ слева от курсора;
Del — стереть символ, на который указывает курсор;
Ctrl-Y — стереть строку, на которой располагается курсор;
Enter — вставить новую строку, разрезать старую;
Ctrl-Q L — восстановить текущую строку (действует, если курсор не покидал измененную строку).

Работа с блоком

Ctrl-K B — пометить начало блока;
Ctrl-K K — пометить конец блока;
Ctrl-K Y — стереть блок;
Ctrl-K C — копировать блок;
Ctrl-K V — переместить блок;
Ctrl-K W — записать блок в дисковый файл;
Ctrl-K R — прочитать блок из дискового файла;
Ctrl-K P — напечатать блок.

1.4. ОСНОВНЫЕ ПРИЕМЫ РАБОТЫ В СРЕДЕ ТУРБО ПАСКАЛЯ

1.4.1. Работа с файлами

Как уже говорилось, сразу после запуска Турбо Паскаля среда автоматически переходит в режим редактирования текста, в котором можно подготовить новую программу или исправить существующую.

Основной формой хранения текстов программ вне среды являются файлы. После завершения работы с Турбо Паскалем можно сохранить текст новой программы в дисковом файле с тем, чтобы использовать его в следующий раз. Для обмена данными между дисковыми файлами и редактором среды предназначены клавиши *F2* (запись в файл) и *F3* (чтение из файла). Если Вы создаете новую программу, то среда еще не знает имя того файла, в который Вы захотите поместить текст этой программы, и поэтому она присваивает тексту стандартное имя *NONAME00.PAS* (*NO NAME — нет имени*). Для сохранения текста программы в файле нужно нажать на клавишу *F2*. В этот момент среда проверит имя и, если это — стандартное имя *NONAME*, спросит, нужно ли его изменять: на экране появится небольшое окно запроса с надписью в верхней части:

Save file as
(Сохранить в файле с именем)

Ниже надписи располагается поле для ввода имени файла, в котором можно написать любое имя и нажать клавишу *Enter*, текст будет сохранен в файле. Если в имени файла опущено расширение, среда присвоит файлу стандартное расширение *.PAS*.

Если завершена работа с Турбо Паскалем (командой *Alt-X*), но не сохранен текст программы на диске, на экране появится окно с запросом:

NONAME00.PAS has been modified.

Save?

(Файл NONAME00.PAS был изменен.

Сохранить?)

В ответ следует нажать *Y* (*Yes* — да), если необходимо сохранить текст в файле, или *N* (*No* — нет), если сохранять текст не нужно.

1.4.2. Прогон и отладка программы

После подготовки текста программы можно попытаться исполнить ее, т.е. откомпилировать программу, связать ее (если необходимо) с библиотекой стандартных процедур и функций, загрузить в оперативную память и передать ей управление. Вся эта последовательность действий называется *прогоном программы* и реализуется командой *Ctrl-F9*.

Если в программе нет синтаксических ошибок, то все действия выполняются последовательно одно за другим, при этом на экране сообщается о количестве строк откомпилированной программы и объеме доступной оперативной памяти. Перед передачей управления загруженной программе среда очищает экран (точнее, выводит на экран окно прогона программы), а после завершения работы программы вновь берет управление компьютером на себя и восстанавливает на экране окно редактора.

Если на каком-либо этапе среда обнаружила ошибку, она прекращает дальнейшие действия, восстанавливает окно редактора и помещает курсор на ту строку программы, при компиляции или исполнении которой обнаружена ошибка. При этом в верхней строке редактора появляется диагностическое сообщение о причине ошибки. Все это позволяет очень быстро отладить программу, т.е. устранить в ней синтаксические ошибки и добиться правильной ее работы.

Если ошибка возникла на этапе работы программы, простое указание того места, где она обнаружена, может не дать нужной информации, так как ошибка может явиться следствием неправильной подготовки данных. Например, если ошибка возникла при извлечении корня из отрицательного числа, будет указан оператор, в котором осуществлялась сама операция извлечения корня, хотя ясно, что первопричину ошибки следует искать где-то раньше, там, где соответствующей переменной присваивается отрицательное значение. В таких ситуациях обычно прибегают к пошаговому исполнению программы с помощью команд, связанных с клавишами *F4*, *F7* и *F8*. Пока еще не накоплен достаточный опыт отладки, можно пользоваться одной клавишей *F7*, после нажатия на которую среда осуществит компиляцию, компоновку (связь с библиотекой стандартных процедур и функций) и загрузку программы, а затем остановит прогон перед исполнением первого оператора. Строка программы, содержащая этот оператор, будет выделена на экране указателем (цветом). Теперь каждое новое нажатие на *F7* будет вызывать исполнение всех операций, запрограммированных в текущей строке, и смещение указателя к сле-

дующей строке программы. В подозрительном месте программы можно просмотреть значения нужных Вам переменных или выражений. Для этого можно действовать следующим образом. Установите курсор в то место текущей строки, где написано имя интересующей Вас переменной, и нажмите *Ctrl-F4*. На экране откроется диалоговое окно, состоящее из трех полей. В верхнем поле будет стоять имя переменной. После этого нажмите на клавишу *Enter*, чтобы получить в среднем поле текущее значение этой переменной. Если перед командой *Ctrl-F4* курсор стоял на пустом участке строки или указывал на другую переменную, верхнее поле также окажется пустым или будет содержать имя этой другой переменной. В этом случае следует ввести с помощью клавиатуры интересующее Вас имя в верхнем поле и нажать клавишу *Enter*. Кстати, таким образом можно вводить не только имена прослеживаемых переменных, но и выражения с их участием — среда вычислит и покажет значение этого выражения.

1.4.3. Справочная Служба Турбо Паскаля

Неотъемлемой составной частью среды Турбо Паскаля является встроенная справочная служба. Если Вы достаточно хорошо владеете английским языком, у Вас не будет проблем при работе с Турбо Паскалем: в затруднительной ситуации достаточно нажать на клавишу *F1* и на экране высветится необходимая справка. Эта справка зависит от текущего состояния среды (такую справочную службу называют контекстно-зависимой). Например, если нажать на *F1* в момент, когда среда обнаружила ошибку в программе, в справке будут сообщены дополнительные сведения о причинах появления этой ошибки и рекомендации по ее устранению.

Существуют четыре способа обращения к справочной службе непосредственно из окна редактора:

F1 — получение контекстно-зависимой справки;

Shift-F1 — выбор справки из списка доступных справочных сообщений;

Ctrl-F1 — получение справки о нужной стандартной процедуре, функции, о стандартной константе или переменной;

Alt-F1 — получение предыдущей справки.

При использовании команды *Shift-F1* на экране появляется справочное окно, содержащее упорядоченный по алфавиту список стандартных процедур, функций, констант и переменных, для которых можно получить справочную информацию. В этот момент клавишами смещения курсора следует передвинуть указатель в окне к нужному слову и нажать клавишу *Enter*, чтобы получить справку.

Эту же справку можно получить и другим способом: напечатать на экране имя стандартной процедуры (функции, константы, переменной) или подвести курсор к имеющемуся уже в тексте программы стандартному имени и нажать *Ctrl-F1*. Среда проанализирует ближайшее окружение курсора, выделит стандартное имя и даст нужную справку.

Доступ к справочной службе возможен и через главное меню Турбо Паскаля (подробнее о работе с меню см. прил.1).

Во многих случаях справка содержит пример небольшой программы, иллюстрирующей соответствующие возможности Турбо Паскаля. Не то — ропитесь запоминать или записывать на бумаге этот текст, его можно «вырезать» из справки и перенести в окно редактора. Для этого после вызова нужной справки нажмите клавишу *Alt* и, не отпуская ее, — клавишу с латинской буквой *E* — на экране раскроется дополнительное меню *Edit*. Затем клавишами смещения курсора подведите указатель (светлый прямоугольник) в меню к строчке *Copy examples* (копировать примеры) и нажмите клавишу *Enter* — текст примера скопируется во внутренний буфер редактора. Для извлечения примера из буфера следует нажать клавишу *Esc*, чтобы выйти из справочной службы, подвести курсор к свободной строке в окне редактора и дать команды *Shift-Ins* (копирование содержимого буфера в виде блока в текст программы) и *Ctrl-K H* (убрать выделение блока цветом).

Глава 2

ЗНАКОМСТВО С ЯЗЫКОМ ТУРБО ПАСКАЛЯ

В этой главе описывается ядро Турбо Паскаля — минимальный набор средств, достаточный для написания сравнительно простых программ. В частности, рассматриваются все операторы языка, наиболее популярные типы данных и операции над ними. Вы познакомитесь с приемами разработки процедур и функций, позволяющими создавать структурированные программы. В заключительной части главы на примерах показано применение методики нисходящего программирования, обеспечивающей сравнительно простой и надежный способ детальной проработки алгоритма программы.

2.1. ВАША ПЕРВАЯ ПРОГРАММА

Для знакомства с языком Турбо Паскаля попробуем составить не — сложную программу, осуществляющую вывод какого — либо сообщения на экран ПК. Пусть это будет фраза «Я программирую на Турбо Паскале». Вот возможный вариант такой программы:

Пример 2.1

```
Program My_First_Program;  
const  
    Text = 'Я программирую на Турбо Паскале';  
begin  
    WriteLn(Text);  
end.
```

Прежде всего проанализируем форму представления текста. В программе шесть строк. Строки программы обычно выделяют некоторые смысловые фрагменты текста и могут не связываться с конкретными действиями в программе: расположение текста программы по строкам — дело вкуса программиста, а не требование синтаксиса языка. Ту же программу можно было бы написать, например, так:

```
Program My_First_Program; const Text =  
'Я программирую на Турбо Паскале';begin WriteLn(Text); end.
```

В отличие от некоторых других языков программирования пробел в языке Турбо Паскаль используется как разделитель отдельных конструкторов языка, поэтому программа

```
PROGRAMMy_First_Program;constText=  
'Я программирую на Турбо Паскале';BEGINWriteLn(Text);end.
```

будет неверной.

В Турбо Паскале игнорируется различие в высоте букв (заглавные или строчные), если только это не связано с текстовыми константами. Начало программы могло бы, например, выглядеть так:

```
program my_first_program;
```

Теперь о смысле отдельных строк. Первая строка

```
Program My_First_Program;
```

начинается словом **Program** и содержит объявление имени программы. Слово **Program** зарезервировано в Турбо Паскале, т.е. не может использоваться ни в каких иных целях, кроме как для объявления имени программы. В Турбо Паскале имеется множество зарезервированных слов (см. гл.3). Любое из них нельзя использовать в качестве идентификатора (имени) какого-либо объекта программы — переменной, константы и т.д. Замечу, что редактор среды Турбо Паскаля обычно выделяет зарезервированные слова цветом. В связи с этим в тексте книги эти слова выделены жирным шрифтом. Поскольку имя программы никак в дальнейшем не используется, требование его объявления кажется излишним. В Турбо Паскале можно опускать объявление имени оператором **Program** без каких-либо последствий для программы.

В рассматриваемом примере имя `My_First_Program` есть не что иное, как английская фраза «Моя Первая Программа», но только написанная без пробелов — пробел является разделителем и не может использоваться произвольно (вместо пробелов в идентификаторах разрешается использовать символ подчеркивания).

Первая строка заканчивается особым разделителем — точкой с запятой. Этот разделитель в языке Турбо Паскаль отмечает конец оператора или описания. Использование особого разделителя позволяет располагать несколько операторов на одной строке.

Вторая строка

```
const
```

содержит единственное зарезервированное слово **const**, означающее, что далее будут описаны одна или несколько констант (*CONSTants* — константы). Константами в языке считаются такие объекты программы, которые не могут изменять своего значения. В отличие от многих других языков программирования, константа в Турбо Паскале может иметь собственное имя, что соответствует принятой в научных и инженерных расчетах практике именования часто используемых констант. Например, со школы мы помним о существовании константы $\pi=3.14159265$. При обработке программы имя константы *pi* будет заменяться компилятором на ее значение.

Описать константу в Турбо Паскале — значит указать ее имя и значение. Такое указание содержится в третьей строке

```
Text = 'Я программирую на Турбо Паскале';
```

в которой константе с именем `Text` присваивается в качестве значения строка символов «Я программирую на Турбо Паскале».

В Турбо Паскале могут использоваться константы разного типа — целые или вещественные числа, символы, строки символов, массивы и т.д. Признаком того, что `Text` является константой типа строка символов, служат два апострофа, обрамляющих строку, причем сами апострофы этой строке не принадлежат, а лишь указывают компилятору на то, что все заключенные в них символы следует рассматривать как единое целое — текстовую константу. Если понадобится включить сам апостроф в текстовую константу, достаточно его написать дважды подряд. Например, описание

```
Text = 'Турбо''Паскаль';
```

создаст константу со значением

```
Турбо'Паскаль
```

Все три первые строки не связаны с какими-либо конкретными действиями при работе программы. Они сообщают компилятору некоторые сведения о самой программе и использующихся в ней объектах. Эта часть программы называется разделом описаний. Зарезервированное слово **begin** в четвертой строке сигнализирует компилятору о начале другой части программы — раздела операторов. В нашем примере этот раздел содержит оператор

```
WriteLn(Text);
```

который, собственно, и выводит сообщение на экран компьютера.

Завершает всю программу зарезервированное слово **end** с точкой. Точка оповещает компилятор о конце текста программы. За сочетанием **end.** можно размещать какой угодно текст — он не будет обрабатываться компилятором.

Перед тем как попробовать откомпилировать и исполнить нашу программу, обсудим ее единственный исполняемый оператор

```
WriteLn(Text);
```

Любопытно, что в Паскале вообще и Турбо Паскале, в частности, нет специальных операторов ввода-вывода. Для обмена информацией с окружающим миром в программах, написанных на языке Турбо Паскаль, используются специальные стандартные процедуры. Таким образом, по своей сути оператор

```
WriteLn(Text);
```

является оператором обращения к встроенной процедуре вывода данных (свое название она получила от *WRITE LiNe* — *записать строку*).

Понятие процедуры (см. гл.8) — одно из центральных понятий Турбо Паскаля. Процедура — это некоторая последовательность операторов, к которой можно обратиться по имени. Всякий раз, когда мы называем в операторе имя процедуры, инициируется последовательность запрограммированных в ней действий.

Процедура `WriteLn` относится к стандартным или встроенным процедурам Турбо Паскаля. Стандартная процедура не нуждается в предварительном описании, она доступна любой программе, в которой содержится обращение к ней. Разница между *оператором вывода* и обращением к *процедуре вывода* состоит в том, что имя процедуры вывода, как и любой другой процедуры Турбо Паскаля, не является зарезервированным словом, а следовательно, пользователь может написать свою собственную процедуру с именем `WriteLn`. Впрочем, эта возможность для большинства пользователей остается лишь языковой тонкостью и очень редко используется на практике.

Процедура `WriteLn` — одна из немногих процедур Турбо Паскаля, при обращении к которым допускается использование произвольного числа параметров. Параметры передаются процедуре в виде списка, располагающегося в круглых скобках сразу за именем процедуры. В нашем примере процедуре передается единственный параметр — константа `Text`. Как мы увидим дальше (см. гл.5), самым первым параметром при обращении к процедуре `WriteLn` можно указать адрес приемника информации — устройство или дисковый файл, в который направляется вывод. Таким способом программист может легко переадресовать вывод данных. Если, как это сделано в нашем примере, адрес вывода не указан, вывод направляется на экран дисплея.

Анализируя всю программу в целом, мы обнаружим, что четыре использованных в ней слова (**Program**, **const**, **begin** и **end**) являются зарезервированными. Слово `WriteLn`, как уже отмечалось, не относится к зарезервированным, но вряд ли может возникнуть необходимость переопределить его, так как в этом случае программа лишится мощного и удобного средства вывода данных. Два слова — `My_First_Program` и `Text` служат идентификаторами (именами) некоторых объектов программы. Программист может использовать в качестве идентификаторов любые последовательности символов, которые удовлетворяют следующим ограничениям:

- идентификатор может состоять из букв латинского алфавита, цифр, знака подчеркивания; никакие другие символы в идентификаторе недопустимы;
- идентификатор не может начинаться с цифры;
- идентификатор не может совпадать ни с одним из зарезервированных слов;
- длина идентификатора может быть произвольной, но значащими считаются первые 63 символа.

Как и всюду в программе, в идентификаторах игнорируется разница в высоте букв, поэтому, например, идентификаторы `Text`, `text` и `TEXT` с точки зрения компилятора идентичны.

Теперь попробуйте выполнить программу. Для этого после набора ее текста нажмите `Ctrl-F9`. Если Вы не ошиблись при вводе текста, то спустя несколько секунд заметите быструю смену изображений на экране: сразу после загрузки программы Турбо Паскаль очищает экран, предоставляя его в распоряжение работающей программы пользователя. Такой экран на —

зывается окном программы. После завершения прогона (работа программы часто называется ее прогоном) на экране вновь появится окно редактора с текстом программы. Если Вы не успели разглядеть изображение окна программы, нажмите *Alt-F5*. После нажатия на любую клавишу среда вернет экран в режим воспроизведения окна редактора.

Перед тем, как двигаться дальше, полезно подробнее ознакомиться с некоторыми возможностями среды Турбо Паскаля. Нажмите клавишу *F10*, чтобы перейти к режиму выбора из главного меню, подведите указатель к опции *Debug* (отладка) и нажмите клавишу *Enter* — на экране раскроется меню второго уровня, связанное с этой опцией. Новое меню как бы «выпало» из верхней строки, поэтому такое меню часто называют выпадающим. Отыщите в новом меню опцию *Output* (вывод программы), подведите к ней указатель и нажмите клавишу *Enter* еще раз. На экране вновь появится окно программы, но оно уже не будет исчезать после нажатия на любую клавишу — экран будет связан с этим окном постоянно. Теперь добьемся того, чтобы на экране демонстрировались два окна одновременно: вновь нажмите клавишу *F10*, выберите *Window*, нажмите клавишу *Enter*, подведите указатель к опции *Tile* (черепица) и нажмите клавишу *Enter* еще раз. Если все сделано правильно, экран приобретет вид, показанный на рис.2.1.

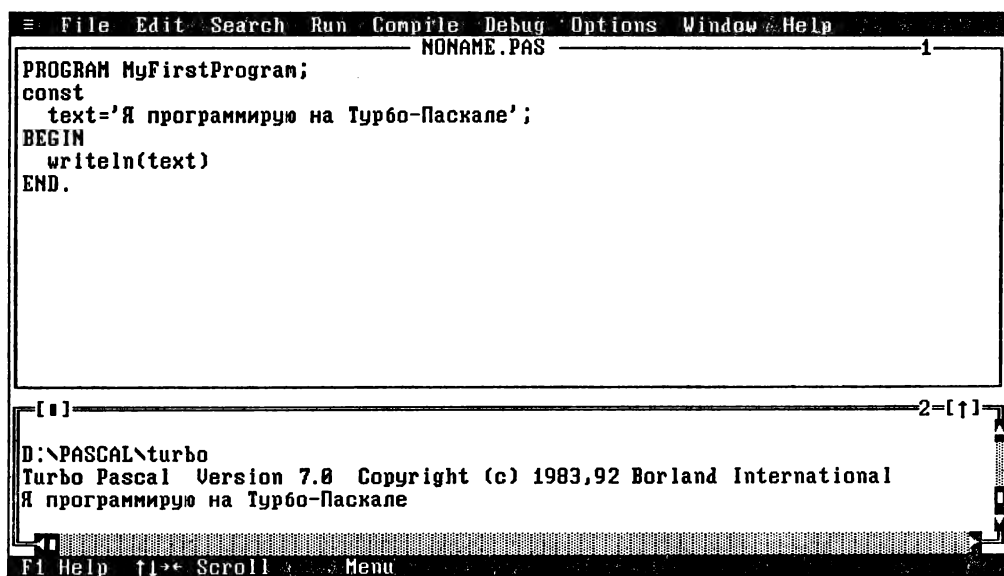


Рис.2.1. Вид экрана с окнами редактора и программы

Двойная рамка, очерчивающая окно программы, свидетельствует о том, что именно это окно активно в данный момент. Сделаем активным окно редактора: нажмем клавишу *Alt* и, не отпуская ее, — клавишу с цифрой 1 (окно редактора имеет номер 1, окно программы — номер 2, см. верхние правые углы рамок на рис.2.1). Теперь все готово к дальнейшим экспериментам с программой.

Попробуем изменить выводимый на экран текст. Например, уберем в конце третьей строки точку с запятой и отредактируем ее таким образом.

```
Text = 'я учусь программировать на турбо паскале'
```

Если Вы запустите программу вновь, нажав клавиши *Ctrl-F9*, компилятор сообщит:

```
Error 85: ";" expected.  
(Ошибка 85: Отсутствует ";").
```

а редактор установит курсор на первый символ слова **begin**, показывая то место, где при разборе текста обнаружена ошибка (разделитель «;» может отделять от конца оператора сколько угодно пробелов; компилятор пропускает эти пробелы в поисках разделителя до тех пор, пока не обнаружит зарезервированное слово, — вот почему курсор стоит не в конце строки с объявлением константы, а перед словом **begin**). Исправьте программу — поставьте в конце третьей строки разделитель «;» и вновь запустите счет. На этот раз все пройдет нормально, но вывод программы изменится, и в окне программы появится текст

```
я учусь программировать на турбо паскале
```

Этот текст строго соответствует заданному в текстовой константе набору символов, поэтому в нем отсутствуют прописные буквы.

2.2. ТИПЫ ДАННЫХ

Структура рассмотренной программы имеет следующий вид:

```
Program MyFirstProgram;  
{Раздел описаний}  
begin  
{Раздел операторов}  
end.
```

Слова **Program**, **begin** и **end** выделяют две части программы — раздел описаний и раздел операторов. Такая структура обязательна для любой программы, что является следствием жесткого требования языка: любой нестандартный идентификатор, используемый в исполняемых операторах, должен быть предварительно описан в разделе описаний. (Стандартные идентификаторы связаны с предварительно объявленными объектами и входят в стандартную библиотеку Турбо Паскаля. Таким, например, является идентификатор `WriteLn`. Стандартные идентификаторы, если они используются в программе, описывать не нужно).

Требование предварительного описания идентификаторов кажется чрезмерно строгим и делающим язык менее свободным. На самом деле в нем проявляется тенденция развития языков программирования в сторону повышения надежности создаваемых программ. Кто программировал на Фортране или Бэйсике (в этих языках не требуется предварительное описание идентификаторов), знает, как порой бывает трудно обнаружить в большой программе ошибочно введенный или пропущенный символ в идентификаторе. Если, например, всюду в программе используется переменная с именем `EPSILON`, а в одном месте ошибочно написано `EPSLON`,

то программа может благополучно откомпилироваться и даже давать почти правдоподобный результат для некоторых наборов данных, но в какой-то момент начнет вести себя странно. Обязательное предварительное описание идентификаторов в Турбо Паскале защищает программы от такого рода ошибок и повышает их надежность.

Описать идентификатор — это значит указать тип связанного с ним объекта программы (константы или переменной). Понятие типа — одно из фундаментальных понятий Турбо Паскаля. В гл.4 подробно рассмотрены различные типы; чтобы пояснить описываемые ниже особенности языка и при этом не слишком забегать вперед, укажем, что тип определяет, во-первых, способ внутреннего для компьютера представления объекта и, во-вторых, действия, которые разрешается над ним выполнять.

В рассматриваемых далее в этой главе программах понадобятся следующие типы данных:

- INTEGER — целочисленные данные, во внутреннем представлении занимают 2 байта; диапазон возможных значений — от -32768 до $+32767$; данные представляются точно;
- REAL — вещественные данные, занимают 6 байт; диапазон возможных значений модуля — от $2.9E-39$ до $1.7E+38$; точность представления данных — 11...12 значащих цифр;
- CHAR — символ, занимает 1 байт;
- STRING — строка символов, занимает $MAX+1$ байт, где MAX — максимальное число символов в строке;
- BOOLEAN — логический тип, занимает 1 байт и имеет два значения: FALSE (ложь) и TRUE (истина).

Тип константы определяется способом записи ее значения. Например:

```
const
  c1 = 17;
  c2 = 3.14;
  c3 = 'A';
  c4 = '3.14';
  c5 = False;
```

При анализе этого фрагмента программы компилятор отнесет первую константу к типу INTEGER, вторую — к типу REAL, третью — к CHAR, четвертую — к STRING и последнюю — к BOOLEAN. Признаком, позволяющим отнести константу к REAL или к INTEGER, является наличие или отсутствие десятичной точки в ее значении. Разумеется, константы C2 и C4 относятся к разным типам: C2 — к REAL (в константе есть десятичная точка), а C4 — к STRING (константа обрамлена апострофами). Константу C3 компилятор будет считать относящейся к типу CHAR: одиночный символ в апострофах относится к CHAR, в то время как несколько символов — к STRING.

В отличие от константы переменная именует объект программы, который может изменять свое значение в ходе счета. При описании переменных за идентификатором ставятся двоеточие и имя типа. Несколько однотипных переменных можно объединять в список, разделяя их запя-

тыми. В начале раздела описания переменных должно стоять зарезервированное слово **VAR** (*VAR*iables — переменные). Например:

```
var
    sigma    : real;
    a,b,c,d  : Char;
    text1    : String[15];
    text2    : String;
    flag     : Boolean;.
```

Как уже говорилось, тип данных определяет длину внутреннего представления соответствующих переменных. В частности, длина внутреннего представления переменных типа **STRING** (строка символов) зависит от максимального числа символов, которые могут составлять строку. В приведенном выше примере переменная *text1* описана с указанием ее максимальной длины (15 символов), а в описании переменной *text2* максимальная длина не указана и компилятор установит для нее предельно допустимую в Турбо Паскале длину — 255 символов.

Рассмотрим еще одну несложную программу (пример 2.2). Ее назначение: ввести с клавиатуры два целых числа, найти результат деления первого числа на второе и вывести полученный результат на экран.

Пример 2.2

```
Program Input_Output;
{Программа вводит два целых числа
 и выводит частное от деления 1-го на 2-е}
var
    n1,n2 : Integer;      {n1 и n2 - вводимые целые}
    x      : real;        {x - результат}
BEGIN
    Write('n1 = ');      {Сообщаем о вводе n1}
    ReadLn(n1);          {Вводим n1}
    Write('n2 = ');      {Сообщаем о вводе n2}
    ReadLn(n2);          {Вводим n2}
    x := n1/n2;          {Находим результат}
    WriteLn('n1/n2 = ',x); {Выводим его}
END.
```

Прежде всего бросается в глаза появление в программе поясняющих комментариев. Комментарий в Турбо Паскале — это произвольная по-следовательность любых символов, обрамленная фигурными скобками. Комментарий разрешается вставлять в любое место программы, где по смыслу может стоять пробел. В качестве ограничителей комментария допускается использование фигурных скобок «{» и «}», а также пары символов: «(» — слева от комментария и «)» — справа от него:

```
    { Это - комментарий }
    (* Это - тоже комментарий *)1
```

¹ Редактор Турбо Паскаля выделяет комментарии наклонным шрифтом (курсивом). Точно так же выделяются комментарии и в тексте книги.

Комментарии с однотипными ограничителями нельзя вкладывать друг в друга, т.е. недопустимы последовательности вида

{ ... { ... } ... } или (* ... (* ... *) ... *)

Однако можно вкладывать комментарии с ограничителями разных типов (не более одной глубины вложения):

{ ... (* ... *) ... } или (* ... { ... } ... *)

Последнее обстоятельство проясняет кажущуюся странной избыточность ограничителей: если всюду в программе будут использоваться ограничители одного типа, то для того, чтобы временно исключить из программы какой-либо фрагмент текста, достаточно заключить его в ограничители другого типа.

Наличие комментариев в программе избавляет меня от необходимости пояснять назначение отдельных строк программы. Несколько слов о вводе данных. Пары операторов

```
Write (...);
ReadLn(...);
```

работают следующим образом. Вначале оператор `Write` выводит строку на экран и оставляет курсор в конце только что выведенной строки текста. Заметим, что оператор

```
WriteLn(Text);
```

в примере 1 после вывода текста осуществлял перевод строки и устанавливал курсор в начало следующей строки экрана. Именно в этом простом действии (переводе строки) заключается единственное отличие в работе процедуры `WriteLn` от процедуры `Write`.

Затем по оператору `ReadLn` вызывается встроенная процедура ввода данных и программа останавливается в ожидании ввода. В этот момент необходимо набрать на клавиатуре нужное число и нажать клавишу *Enter*. Сразу после этого программа продолжит работу: проанализирует введенное число и перейдет к вводу следующего числа или вычислению результата. Таким образом, сигналом окончания подготовки очередного числа является нажатие на клавишу *Enter*, до этого момента можно стирать любой ошибочно введенный символ клавишей *Backspace*.

Для вычисления отношения введенных чисел используется один из основных операторов Турбо Паскаля — оператор присваивания. В его левой части указывается имя переменной, правая часть представляет собой выражение того же типа, что и переменная. Пара символов «:=», связывающая левую и правую части оператора присваивания, означает «присвоить значение». Запомним: в операторах присваивания Турбо Паскаля всегда используются символы «:=», в то время как при описании констант — одиночный символ «=». С точки зрения синтаксиса языка, два символа «:=» рассматриваются как один специальный символ и обязательно пишутся слитно.

Оператор присваивания используется практически во всех языках программирования. В некоторых языках, например в Фортране или Бейсике, символом присваивания является знак равенства, однако новичка, привыкшего к строгости математических формул, может озадачить типичная форма записи фортран-оператора присваивания, например, такая:

$$X = X + 1$$

Вариант записи этого же оператора на Турбо Паскале:

$$X := X + 1;$$

в этом смысле кажется более логичным. Разумеется, вряд ли кому-нибудь придет в голову видеть уравнения там, где их нет и не может быть. Конечно же, и в том, и в другом случае реализуется одно и то же алгоритмическое действие: к содержимому X прибавляется 1 и полученный результат вновь присваивается переменной X . Обратите внимание на оператор вывода результатов

$$\text{WriteLn}('n1/n2 = ', x);$$

В нем в качестве одного из параметров явно указывается константа типа строка символов `'n1/n2 = '`. Конечно же, константы (в отличие от переменных) вовсе не обязательно описывать в разделе описаний, так как их тип легко определяется компилятором по форме записи константы. С учетом этого можно было бы записать программу из примера 1 предельно лаконично:

```
begin WriteLn('Я программирую на Турбо Паскале'); end.
```

2.3. ПРЕОБРАЗОВАНИЯ ТИПОВ И ДЕЙСТВИЯ НАД НИМИ

Как уже говорилось, тип переменной позволяет не только устанавливать длину ее внутреннего представления, но и контролировать те действия, которые выполняются над ней в программе. Контроль за использованием переменных еще на этапе компиляции программы — важное преимущество Турбо Паскаля перед другими языками программирования, в которых допускается автоматическое преобразование типов. В Турбо Паскале почти невозможны неявные (автоматические) преобразования типов. Исключение сделано только в отношении констант и переменных типа `INTEGER` (целые), которые разрешается использовать в выражениях типа `REAL` (вещественные). Если, например, переменные X и Y описаны следующим образом:

```
var
  x: Integer;
  y: real;
```

то оператор

$$y := x + 2;$$

будет синтаксически правильным: хотя справа от знака присваивания стоит целочисленное выражение, а слева — вещественная переменная,

компилятор сделает необходимые преобразования автоматически. В то же время оператор

```
x := 2.0;
```

будет неверным, так как автоматическое преобразование типа REAL (константа 2.0 содержит десятичную точку и, следовательно, принадлежит к типу REAL) в тип INTEGER в Турбо Паскале запрещено.

Разумеется, запрет на автоматическое преобразование типов еще не означает, что в Турбо Паскале нет средств преобразования данных. Они, конечно же, есть, но их нужно использовать явно (подробнее об этом см. гл.4). Для преобразования данных в языке существуют встроенные функции, которые получают в качестве параметра значение одного типа, а возвращают результат в виде значения другого типа. В частности, для преобразования REAL в INTEGER имеются даже две встроенные функции такого рода: ROUND округляет REAL до ближайшего целого, а TRUNC усекает REAL путем отбрасывания дробной части.

Например, ошибочным будет оператор

```
x := y/x;
```

но правильным

```
x := round(y/x);
```

(объявления переменных см. выше).

Понятие функции в Турбо Паскале близко к понятию процедуры. Как и процедура, функция вызывается своим именем и может содержать произвольное число операторов Турбо Паскаля и даже внутренних процедур и функций. Существенным отличием функции от процедуры является то обстоятельство, что функция имеет собственное значение и, следовательно, может использоваться наравне с переменными в выражениях соответствующего типа.

Для преобразования данных типа CHAR (символ) в целое число предназначена функция ORD, обратное преобразование INTEGER в CHAR осуществляет функция CHR.

С помощью следующей несложной программы (пример 2.3) Вы сможете узнать внутренний код произвольного символа.

Пример 2.3

Program Code_of_Char;

{Программа читает символ с клавиатуры и выводит на экран этот символ и соответствующий ему внутренний код}

var

ch: Char; {В эту переменную читается символ}

begin

Write('Введите любой символ: ');

ReadLn(ch); {Читаем один символ}

WriteLn(ch, ' = ', ord(ch)); {Преобразуем его к целому и выводим на экран}

END.

Обратите внимание: при вызове

```
WriteLn(ch, ' = ', ord(ch));
```

третьим параметром обращения указан вызов функции `ORD(CH)`, что с точки зрения языка является *выражением*; как мы увидим дальше (см. гл.8), во многих случаях при вызове процедур и функций в качестве параметров вызова можно указывать не только переменные или константы, но и выражения с их участием.

По мере надобности мы будем знакомиться с другими функциями преобразования типов данных, а сейчас — о тех операциях, которые разрешены над различными типами.

Конечно же, в Турбо Паскале есть все четыре арифметические операции над переменными `REAL` и `INTEGER`:

+	— сложение;
-	— вычитание;
*	— умножение;
/	— деление вещественное;
div	— деление целочисленное.

Наличие двух операций деления есть еще одно проявление основополагающего принципа Турбо Паскаля: программист должен явно подтверждать компилятору, что он готов к возможным последствиям преобразования типов. Если, например, в языке Фортран используется выражение $1/2$, то результат этого выражения будет зависеть от того, переменной какого типа он будет присвоен: если N есть переменная целого типа, а X — вещественного, то в программе на Фортране присваивания

```
N = 1/2  
X = 1/2
```

дадут значения 0 для N и 0.5 для X . В Турбо Паскале такой двусмысленности нет: выражение $1/2$ всегда имеет значение 0.5 и поэтому оператор

```
var  
  N : Integer;  
begin  
  N := 1/2;
```

просто недопустим. В то же время допустимый в Турбо Паскале оператор

```
var  
  X : real;  
begin  
  X := 1 div 2;
```

самим фактом использования операции целочисленного деления `DIV` свидетельствует о том, что программист сознательно отбрасывает дробную часть результата. (Надеюсь, что читатель извинит явную искусственность

этих примеров, которая вызвана лишь стремлением проиллюстрировать обсуждаемые особенности языка).

Для данных типа **INTEGER** в Турбо Паскале есть еще одна операция **MOD** — получение остатка от целочисленного деления. Например:

```
5 mod 2      = 1
31 mod 16    = 15
18 mod 3     = 0
```

В Турбо Паскале отсутствует операция возведения в степень, что, очевидно, будет вызывать определенные неудобства при реализации вычислительных алгоритмов. Некоторым утешением может служить наличие встроенной функции **SQR**, возвращающей квадрат от значения параметра, причем тип результата определяется типом параметра.

И еще об одном существенном недостатке Турбо Паскаля: в нем отсутствуют комплексный тип и соответствующие операции над ним. Вообще, в отношении реализации разнообразных вычислительных процедур Турбо Паскаль значительно уступает некоторым другим языкам программирования, в частности, тому же Фортрану. В частности, в нем на много беднее набор встроенных математических функций (см. гл. 4).

При работе с целыми числами могут оказаться полезными две процедуры (здесь и далее в квадратных скобках указываются необязательные параметры):

DEC(**X** [, **N**]) — уменьшает содержимое переменной **X** на значение выражения **N** (если **N** не задано, то на 1); тип переменной **X** и выражения **N** — **INTEGER** (точнее, любой целый, см. гл. 4);

INC(**X** [, **N**]) — увеличивает значение **X** на **N** (если **N** не задано, то на 1).

Над символами и строками символов определена единственная операция — сцепление двух строк. Операция обозначается символом «+». Например, программа

```
var
  st: String;
begin
  st := 'Турбо'+'-'+'Паскаль';
  WriteLn(st);
end.
```

напечатает строку

Турбо-Паскаль

Все остальные действия над строками и символами реализуются с помощью встроенных процедур и функций (см. гл. 4).

И, наконец, об операциях отношения и логических операциях.

Над данными типа **REAL**, **INTEGER**, **CHAR**, **STRING** определены следующие операции отношения (сравнения):

```
=      — равно;
<>     — не равно;
<      — меньше;
```

> — больше;
 <= — меньше или равно,
 >= — больше или равно.

В операциях сравнения должны участвовать однотипные операнды. Исключение сделано опять — таки в отношении REAL и INTEGER, которые могут сравниваться друг с другом. Результат применения операции отношения к любым операндам имеет тип BOOLEAN.

Сравнение двух строк осуществляется следующим образом. Символы строк сравниваются попарно друг с другом так, что первый символ первой строки сравнивается с первым символом второй строки, второй символ первой строки — со вторым символом второй и т.д. Символы сравниваются путем сравнения их кодов во внутреннем представлении (см. гл. 4). Если одна строка короче другой, недостающие символы заменяются нулем. Отношение первой несовпадающей друг с другом пары символов и принимается за отношение двух строк.

При сравнении данных типа BOOLEAN учитывается внутреннее соглашение Турбо Паскаля, в соответствии с которым FALSE есть нулевой байт, а TRUE — байт с единицей в младшем разряде. Заметим, что функция ORD преобразует к целому не только символы, но и логические величины, поэтому

```
ord(false) = 0,
ord(true)  = 1.
```

В Турбо Паскале определены следующие логические операции:

not — логическое НЕ; **or** — логическое ИЛИ;
and — логическое И; **xor** — исключающее ИЛИ.

Логические операции применимы к операндам целого и логического типов. Если операнды — целые числа, то результат логической операции есть тоже целое число (подробнее об этом сказано в гл.4). Логические операции над логическими данными дают результат логического типа.

При вычислении выражений любого типа приоритет вычислений определяется расставленными скобками, а при их отсутствии — по табл. 2.1 (в порядке убывания приоритета).

Таблица 2.1

Приоритет операций

Приоритет	Операция
1	not , @
2	* , / , div , mod , and , shl , shr
3	+ , - , or , xor
4	= , <> , > , >= , < , <= , in

Примечание. Операции @ (получение адреса), **shl** (сдвиг влево), **shr** (сдвиг вправо) и **in** (принадлежность к множеству) описаны в гл.4.

Следует учесть, что в отличие от многих других языков программирования в Турбо Паскале логические операции имеют более высокий при-

оритет, чем операции отношения. В связи с этим, в сложных логических выражениях обычно необходимо расставлять скобки. Если, например, *b* и *c* имеют тип `INTEGER`, то выражение

`a = b and c < d`

вызовет сообщение о синтаксической ошибке, так как сначала выполнится операция `b and c`. Правильным будет выражение:

`(a = b) and (c < d)`

2.4. ОПЕРАТОРЫ ЯЗЫКА

С одним из наиболее часто используемых операторов языка Турбо Паскаль — оператором присваивания мы уже познакомились. Ниже рассматриваются остальные операторы языка.

2.4.1. Составной оператор и пустой оператор

Составной оператор — это последовательность произвольных операторов программы, заключенная в операторные скобки — зарезервированные слова `begin ... end`. Составные операторы — важный инструмент Турбо Паскаля, дающий возможность писать программы по современной технологии структурного программирования (без операторов перехода `GOTO`).

Язык Турбо Паскаль не накладывает никаких ограничений на характер операторов, входящих в составной оператор. Среди них могут быть и другие составные операторы — Турбо Паскаль допускает произвольную глубину их вложенности:

```
begin
  .....
  begin
    .....
    begin
      .....
      .....
    end;
    .....
  end;
  .....
end;
```

Фактически, весь раздел операторов, обрамленный словами `begin ... end`, представляет собой один составной оператор. Поскольку зарезервированное слово `end` является закрывающей операторной скобкой, оно одновременно указывает и конец предыдущего оператора, поэтому ставить перед ним символ «;» необязательно, и далее во всех примерах мы не будем этого делать. Наличие точки с запятой перед `end` в предыдущих примерах означало, что между последним оператором и операторной скобкой `end` располагается *пустой оператор*. Пустой оператор не со-

держит никаких действий, просто в программу добавляется лишняя точка с запятой. В основном пустой оператор используется для передачи управления в конец составного оператора.

2.4.2. Условный оператор

Условный оператор позволяет проверить некоторое условие и в зависимости от результатов проверки выполнить то или иное действие. Таким образом, условный оператор — это средство ветвления вычислительного процесса.

Структура условного оператора имеет следующий вид:

```
IF <условие> THEN <оператор1> ELSE <оператор2> ,
```

где **IF**, **THEN**, **ELSE** — зарезервированные слова (если, то, иначе);

<условие> — произвольное выражение логического типа;

<оператор1>, <оператор2> — любые операторы языка Турбо Паскаль.

Условный оператор работает по следующему алгоритму. Вначале вычисляется условное выражение <условие>. Если результат есть TRUE (истина), то выполняется <оператор1>, а <оператор2> пропускается; если результат есть FALSE (ложь), наоборот, <оператор1> пропускается, а выполняется <оператор2>. Например:

```
var
  x, y, max: Integer;
begin
  .....
  if x > max then
    y := max
  else
    y := x;
```

При выполнении этого фрагмента переменная Y получит значение переменной X, если только это значение не превышает MAX, в противном случае Y станет равно MAX.

Часть **ELSE** <оператор2> условного оператора может быть опущена. Тогда при значении TRUE условного выражения выполняется <оператор1>, в противном случае этот оператор пропускается:

```
var
  x, y, max: Integer;
begin
  .....
  if x > max then
    max := x;
  y := x;
```

В этом примере переменная Y всегда будет иметь значение переменной X, а в MAX запоминается максимальное значение X.

Поскольку любой из операторов <оператор1> и <оператор2> может быть любого типа, в том числе и условным, а в то же время не каждый из «вложенных» условных операторов может иметь часть **ELSE** <оператор2>, то возникает неоднозначность трактовки условий. Эта неоднозначность в Турбо Паскале решается следующим образом: любая встретившаяся часть **ELSE** соответствует ближайшей к ней «сверху» части **THEN** условного оператора. Например:

```
var
  a,b,c,d : Integer;
begin
  a := 1; b := 2; c := 3; d := 4;
  if a > b then
    if c < d then
      if c < 0 then
        c := 0
      else
        a := b; {a равно 1}
    if a > b then
      if c then
        if c then
          c := 0
        else
          else
        else
      else
    a := b; {a равно 2}
```

Рассмотрим программу (пример 2.4), которая вводит произвольное десятичное целое число в диапазоне 0...15, преобразует его к шестнадцатеричному и выводит на экран полученный результат.

Пример 2.4

Program Hex;

{Программа вводит с клавиатуры целое число в диапазоне от 0 до 15, преобразует его к шестнадцатеричной системе счисления и выводит результат на экран}

var

 n : Integer; {Вводимое число}
 ch : Char; {Результат}

begin

 Write('n = ');

 ReadLn(n); { Вводим число }

 {Проверяем число на принадлежность к диапазону 0...15}

if (n >= 0) **and** (n <= 15) **then**

begin {Да, принадлежит диапазону}

if n < 10 **then**

 ch := chr(ord('0') + n)

else

 ch := chr(ord('A') + n - 10);

```

      WriteLn('n = ',ch)
    end
  else                                     {Не принадлежит диапазону}
    WriteLn('Ошибка')
  end.

```

В шестнадцатеричной системе счисления используется 16 цифр в каждом разряде: цифры 0...9 обозначают первые 10 возможных значений разряда, буквы A...F — остальные шесть.

В программе учитывается непрерывность и упорядоченность множеств цифр 0...9, букв A...F и их кодов (см. гл.4).

2.4.3. Операторы повторений

В языке Турбо Паскаль имеются три различных оператора, с помощью которых можно запрограммировать повторяющиеся фрагменты программ.

Счетный оператор цикла FOR имеет такую структуру:

```
FOR <пар_цик> := <нач_знач> TO <кон_знач> DO <оператор>.
```

Здесь **FOR**, **TO**, **DO** — зарезервированные слова (*для*, *до*, *выполнить*);

<пар_цик> — параметр цикла — переменная типа INTEGER (точнее, любого порядкового типа, см. гл.4);

<нач_знач> — начальное значение — выражение того же типа;

<кон_знач> — конечное значение — выражение того же типа;

<оператор> — произвольный оператор Турбо Паскаля.

При выполнении оператора **FOR** вначале вычисляется выражение <нач_знач> и осуществляется присваивание <пар_цик> := <нач_знач>. После этого циклически повторяется:

- проверка условия <пар_цик> <= <кон_знач>; если условие не выполнено, оператор **FOR** завершает свою работу;
- выполнение оператора <оператор>;
- наращивание переменной <пар_цик> на единицу.

В качестве иллюстрации применения оператора **FOR** рассмотрим программу, осуществляющую ввод с клавиатуры произвольного целого числа N и вычисление суммы всех целых чисел от 1 до N (пример 2.5).

Пример 2.5

```
Program Summ_of_Integer;
```

{Программа вводит целое положительное число N и подсчитывает сумму всех целых чисел от 1 до N}

```
var
```

```
  i, n, s : Integer;
```

```
begin
```

```
  Write('N = ');
```

```
  ReadLn(n);
```

{Вводим N}

```
  s := 0;
```

{Начальное значение суммы}

```
  for i := 1 to n do
```

{Цикл подсчета суммы}

```

    s := s + i;
    writeln('Сумма = ', s)      {Выводим результат}
end.

```

Отметим два обстоятельства. Во-первых, условие, управляющее работой оператора **FOR**, проверяется перед выполнением оператора <оператор>: если условие не выполняется в самом начале работы оператора **FOR**, исполняемый оператор не будет выполнен ни разу. Другое обстоятельство — шаг наращивания параметра цикла строго постоянен и равен (+1). Существует другая форма оператора:

```

FOR <пар_цикл> := <нач_знач> DOWNTO <кон_знач> DO <оператор>

```

Замена зарезервированного слова **TO** на **DOWNTO** означает, что шаг наращивания параметра цикла равен (−1), а управляющее условие приобретает вид <пар_цикл> = <кон_знач>.

Пример 2.5 можно модифицировать так, чтобы сделать его пригодным для подсчета любых сумм — положительных и отрицательных:

```

.....
s := 0;
if n >= 0 then
    for i := 1 to n do
        s := s + i
    else
        for i := -1 downto n do
            s := s + i;
        .....

```

Два других оператора повторений лишь проверяют условие выполнения или повторения цикла, но не связаны с изменением счетчика цикла.

Оператор цикла WHILE с предпроверкой условия:

```

WHILE <условие> DO <оператор>.

```

Здесь **WHILE**, **DO** — зарезервированные слова (пока [выполняется условие], делать);

<условие> — выражение логического типа;

<оператор> — произвольный оператор Турбо Паскаля.

Если выражение <условие> имеет значение **TRUE**, то выполняется <оператор>, после чего вычисление выражения <условие> и его проверка повторяются. Если <условие> имеет значение **FALSE**, оператор **WHILE** прекращает свою работу.

Рассмотрим пример 2.6, иллюстрирующий использование оператора **WHILE**. Найдем так называемое «машинное epsilon» — такое минимальное, не равное нулю вещественное число, которое после прибавления его к 1.0 еще дает результат, отличный от 1.0.

Пример 2.6

Program EpsilonDetect;

```
{Программа вычисляет и выводит на экран
значение "машинного эпсилон"}
var
  epsilon: real;
begin
  epsilon := 1;
  while epsilon/2 + 1 > 1 do
    epsilon := epsilon/2
  WriteLn('Машинное эпсилон = ',epsilon)
end.
```

У читателя, привыкшего к непрерывной вещественной арифметике, может вызвать недоумение утверждение о том, что в дискретной машинной арифметике всегда существуют такие числа $0 < X < \epsilon$, что $1.0 + X = 1.0$. Дело в том, что внутреннее представление типа REAL может дать «лишь» приблизительно 10^{14} возможных комбинаций значащих разрядов в отведенных для него 6 байтах. Конечно же, это очень большое число, но оно несопоставимо с бесконечным множеством вещественных чисел. Аппроксимация бесконечного непрерывного множества вещественных чисел конечным (пусть даже и очень большим) множеством их внутреннего машинного представления и приводит к появлению «машинного эпсилон».

Оператор цикла REPEAT... UNTIL с постпроверкой условия:

REPEAT <тело_цикла> **UNTIL** <условие>.

Здесь **REPEAT**, **UNTIL** — зарезервированные слова (*повторять до тех пор, пока не будет выполнено условие*);

<тело_цикла> — произвольная последовательность операторов Турбо Паскаля;

<условие> — выражение логического типа.

Операторы <тело_цикла> выполняются хотя бы один раз, после чего вычисляется выражение <условие>: если его значение есть FALSE, операторы <тело_цикла> повторяются, в противном случае оператор **REPEAT... UNTIL** завершает свою работу.

Для иллюстрации применения оператора **REPEAT... UNTIL** модифицируем программу из примера 2.3. Модификация (пример 2.7) состоит в том, что программа будет все время повторять цикл ввода символа и печати его кода до тех пор, пока очередным символом не будет символ CR (вводится клавишей Enter).

Пример 2.7

Program Codes_of_Chars;

{Программа вводит символ и выводит на экран его код. Для завершения работы программы нужно дважды нажать Enter}

var

ch : Char; {Вводимый символ}

const

```

CR = 13;           {Код символа CR}
begin
  repeat
    ReadLn(ch);
    WriteLn(ch, ' = ', ord(ch))
  until ord(ch) = CR
end.

```

Обратите внимание: пара **REPEAT... UNTIL** подобна операторным скобкам **begin... end**, поэтому перед **UNTIL** ставить точку с запятой необязательно.

Для гибкого управления циклическими операторами **FOR**, **WHILE** и **REPEAT** в состав Турбо Паскаля включены две процедуры:

BREAK — реализует немедленный выход из цикла; действие процедуры заключается в передаче управления оператору, стоящему сразу за концом циклического оператора;

CONTINUE — обеспечивает досрочное завершение очередного прохода цикла; эквивалент передачи управления в самый конец циклического оператора.

Введение в язык этих процедур практически исключает необходимость использования операторов безусловного перехода **GOTO** (см. ниже п.2.4.5).

2.4.4. Оператор выбора

Оператор выбора позволяет выбрать одно из нескольких возможных продолжений программы. Параметром, по которому осуществляется выбор, служит *ключ выбора* — выражение любого порядкового типа (любого из рассмотренных, кроме типов **REAL** и **STRING**, см. гл. 4).

Структура оператора выбора такова:

```
CASE <ключ_выбора> OF <список_выбора> [ELSE <операторы>] END
```

Здесь **CASE**, **OF**, **ELSE**, **END** — зарезервированные слова (*случай, из, иначе, конец*);

<ключ_выбора> — ключ выбора;

<список_выбора> — одна или более конструкций вида:

<константа_выбора> : <оператор>;

<константа_выбора> — константа того же типа, что и выражение <ключ_выбора>;

<операторы> — произвольные операторы Турбо Паскаля.

Оператор выбора работает следующим образом. Вначале вычисляется значение выражения <ключ_выбора>, а затем в последовательности операторов <список_выбора> отыскивается такой, которому предшествует константа, равная вычисленному значению. Найденный оператор выполняется, после чего оператор выбора завершает свою работу. Если в списке выбора не будет найдена константа, соответствующая вычисленному значению ключа выбора, управление передается операторам, стоящим за

словом **ELSE**. Часть **ELSE** <оператор> можно опускать. Тогда при отсутствии в списке выбора нужной константы ничего не произойдет и оператор выбора просто завершит свою работу.

Составим программу (пример 2.8), имитирующую работу микрокалькулятора. Программа вводит две строки: первая содержит два произвольных числа, разделенных пробелом, вторая — символ арифметического действия, например:

2 2
*

или

18.35 0.12
/

Над введенными числами осуществляется соответствующее действие и результат выводится на экран. Признаком конца работы программы служит ввод любого символа, отличного от +, -, *, /.

Пример 2.8

Program Calc;

{Программа вводит два числа в первой строке и один из знаков +, -, *, / - во второй и выводит на экран результат соответствующего арифметического действия}

var
 operation : Char; {Знак операции}
 x, y, z : real; {Операнды и результат}
 stop : Boolean; {Признак ошибочной операции
 и останова}

begin

 stop := false;

repeat

 WriteLn; {Пустая строка-разделитель}

 Write('x,y= ');

 ReadLn(x,y);

 Write('операция: ');

 ReadLn(operation);

case operation **of**

 '+' : z := x + y;

 '-' : z := x - y;

 '*' : z := x * y;

 '/' : z := x / y;

else

 stop := true;

end;

if not stop **then**

 WriteLn(' результат= ',z)

until stop

end.

Любому из операторов списка выбора может предшествовать не одна, а несколько констант выбора, разделенных запятыми. Например, следующая

программа при вводе одного из символов: *y* или *Y* выведет на экран слово «Да», а при вводе *n* или *N* — слово «Нет»:

```
var
  ch : Char;
begin
  ReadLn(ch);
  case ch of
    'n', 'N': WriteLn('Нет');
    'y', 'Y': WriteLn('Да')
  end
end.
```

2.4.5. Метки и операторы перехода

Можно теоретически показать, что рассмотренных операторов вполне достаточно для написания программ любой сложности. В этом отношении наличие в языке операторов перехода кажется излишним. Более того, современная технология структурного программирования основана на принципе «программировать без GOTO»: считается, что злоупотребление операторами перехода затрудняет понимание программы, делает ее запутанной и сложной в отладке.

Тем не менее, в некоторых случаях использование операторов перехода может упростить программу.

Оператор перехода имеет вид:

GOTO <метка>.

Здесь **GOTO** — зарезервированное слово (*перейти [на метку]*);
<метка> — метка.

Метка в Турбо Паскале — это произвольный идентификатор, позволяющий именовать некоторый оператор программы и таким образом ссылаться на него. В целях совместимости со стандартным языком Паскаль в языке Турбо Паскаль допускается в качестве меток использование также целых чисел без знака.

Метка располагается непосредственно перед помечаемым оператором и отделяется от него двоеточием. Оператор можно пометить несколькими метками, которые в этом случае отделяются друг от друга двоеточием. Перед тем как появиться в программе, метка должна быть описана. Описание меток состоит из зарезервированного слова **LABEL** (*метка*), за которым следует список меток:

```
label
  loop, lb1, lb2;
begin
  .....
  goto lb1;
  .....
  loop: .....
  .....
```



```
lb1:lb2: .....  
.....  
goto lb2;  
.....
```

Действие оператора **GOTO** состоит в передаче управления соответствующему меченному оператору.

При использовании меток необходимо руководствоваться следующими правилами:

- метка, на которую ссылается оператор **GOTO**, должна быть описана в разделе описаний и она обязательно должна встретиться где-нибудь в теле программы;
- метки, описанные в процедуре (функции), локализируются в ней, поэтому передача управления извне процедуры (функции) на метку внутри нее невозможна.

2.5. МАССИВЫ

Рассмотренные выше простые типы данных позволяют использовать в программе одиночные объекты — числа, символы, строки и т.п. В Турбо Паскале могут использоваться также объекты, содержащие множество однотипных элементов. Это *массивы* — формальное объединение нескольких однотипных объектов (чисел, символов, строк и т.п.), рассматриваемое как единое целое. К необходимости применения массивов мы приходим всякий раз, когда требуется связать и использовать целый ряд родственных величин. Например, результаты многократных замеров температуры воздуха в течение года удобно рассматривать как совокупность вещественных чисел, объединенных в один сложный объект — массив измерений.

При описании массива необходимо указать общее число входящих в массив элементов и тип этих элементов. Например:

```
var  
  a : array [1..10] of real;  
  b : array [0..50] of Char;  
  c : array [-3..4] of Boolean;
```

Как видим, при описании массива используются зарезервированные слова **ARRAY** и **OF** (*массив, из*). За словом **ARRAY** в квадратных скобках указывается тип-диапазон, с помощью которого компилятор определяет общее число элементов массива. Тип-диапазон (подробнее см. в гл.4) задается левой и правой границами изменения индекса массива, так что массив *A* состоит из 10 элементов, массив *B* — из 51, а массив *C* — из 8 элементов. За словом **OF** указывается тип элементов, образующих массив.

Доступ к каждому элементу массива в программе осуществляется с помощью *индекса* — целого числа (точнее, выражения порядкового типа, см. гл.4), служащего своеобразным именем элемента в массиве (если левая граница типа-диапазона равна 1, индекс элемента совпадает с его порядковым номером). При упоминании в программе любого элемента

массива сразу за именем массива должен следовать индекс элемента в квадратных скобках, например:

```
var
  a: array [1..10] of Integer;
  b: array [0..40] of Char;
  c: array [-2..2] of Boolean;
  k: Integer;
begin
  b[17] := 'F';
  c[-2] := a[1] > [2];
  for k := 1 to 10 do
    a[k] := 0;
  ...
end.
```

В правильно составленной программе индекс не должен выходить за пределы, определенные типом — диапазоном. Например, можно использовать элементы $A[1]$, $B[38]$, $C[0]$, но нельзя $A[0]$ или $C[38]$ (определение массивов см. выше). Турбо Паскаль может контролировать использование индексов в программе на этапе компиляции и на этапе счета программы.

Для иллюстрации приемов работы с массивами составим программу (пример 2.9), которая создает массив случайных целых чисел, подсчитывает их среднее арифметическое, а также определяет и выводит на экран минимальное и максимальное из этих чисел.

Пример 2.9

Program Average;

{Программа создает массив из N случайных целых чисел, равномерно распределенных в диапазоне от 0 до MAX_VALUE-1, подсчитывает среднее арифметическое этих чисел, а также минимальное и максимальное из них.}

const

N = 1000; {Количество элементов массива}
MAX_VALUE = 100+1; {Диапазон значений случайных чисел}

var

m : array [1..N] of Integer; {Массив чисел}
i : Integer; {Индекс массива}
max, min : Integer; {Максимальное и минимальное число}
s : real; {Сумма чисел}

begin

{Наполняем массив случайными числами:}

for i := 1 to N do
 m[i] := random(MAX_VALUE);

{Задаем начальные значения переменных:}

s := 0;
max := m[1];
min := m[1];

{Цикл вычисления суммы всех случайных чисел и поиска

```
минимального и максимального:}
for i := 1 to N do
begin
  s := s + m[i];
  if m[i] < min then
    min := m[i]
  else if m[i] > max then
    max := m[i]
end;
{Вычисляем среднее значение и печатаем результат:}
WriteLn('Мин = ', min, ' Макс = ', max, ' Среднее = ', s/N)
end.
```

Для создания массива используется встроенная функция `RANDOM (MAX)`, которая возвращает случайное целое число, равномерно распределенное в диапазоне от 0 до `MAX-1` (`MAX` — параметр обращения).

2.6. ПРОЦЕДУРЫ И ФУНКЦИИ

Процедуры и функции представляют собой важный инструмент Турбо Паскаля, позволяющий писать хорошо структурированные программы. В структурированных программах обычно легко прослеживается основной алгоритм, их нетрудно понять любому читателю, они проще в отладке и менее чувствительны к ошибкам программирования. Все эти свойства являются следствием важной особенности процедур (функций), каждая из которых представляет собой во многом самостоятельный фрагмент программы, связанный с основной программой лишь с помощью нескольких параметров. Самостоятельность процедур (функций) позволяет локализовать в них все детали программной реализации того или иного алгоритмического действия и поэтому изменение этих деталей, например, в процессе отладки обычно не приводит к изменениям основной программы.

Многие примеры в этой книге невелики по размерам (не более 30 — 40 строк), поэтому написать такие программы можно и без процедур. Иное дело — создание крупных программ в сотни, тысячи и десятки тысяч строк. Писать такие программы как нечто единое целое, без расчленения на относительно самостоятельные фрагменты, т.е. без *структурирования*, просто невозможно. Практически во всех языках программирования имеются средства структурирования. Языки, в которых предусмотрены такие механизмы, называются *процедурно-ориентированными*. К их числу принадлежит и Турбо Паскаль.

Процедурой в Турбо Паскале называется особым образом оформленный фрагмент программы, имеющий собственное имя. Упоминание этого имени в тексте программы приводит к активизации процедуры и называется ее *вызовом*. Сразу после активизации процедуры начинают выполняться входящие в нее операторы, после выполнения последнего из них управление возвращается обратно в основную программу и выполняются операторы, стоящие непосредственно за оператором вызова процедуры (рис.2.2).

Для обмена информацией между основной программой и процедурой используется один или несколько параметров вызова. Как мы увидим дальше (см. гл. 8), процедуры могут иметь и другой механизм обмена данными с вызывающей программой, так что параметры вызова могут и не использоваться. Если они есть, то они перечисляются в круглых скобках за именем процедуры и вместе с ним образуют оператор вызова процедуры.

Функция отличается от процедуры тем, что результат ее работы возвращается в виде значения этой функции, и, следовательно, вызов функции может использоваться наряду с другими операндами в выражениях.

С примерами процедур и функций мы уже сталкивались — это стандартные процедуры чтения и записи READ, READLN, WRITE, WRITELN, функции ORD, CHR, математические функции и др. Стандартными они называются потому, что созданы одновременно с системой Турбо Паскаль и являются ее неотъемлемой частью. В Турбо Паскале имеется много стандартных процедур и функций. Наличие богатой библиотеки таких программных заготовок существенно облегчает разработку прикладных программ. Однако в большинстве случаев некоторые специфичные для данной прикладной программы действия не находят прямых аналогов в библиотеках Турбо Паскаля, и тогда программисту приходится разрабатывать свои, нестандартные процедуры и функции.

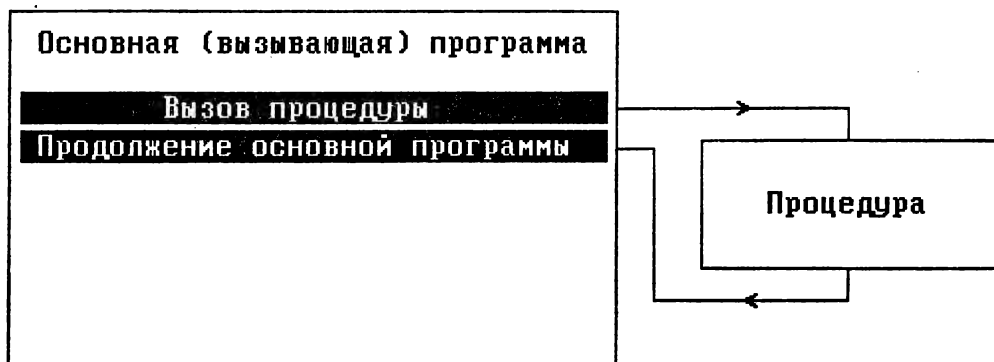


Рис.2.2. Взаимодействие вызывающей программы и процедуры

Нестандартные процедуры и функции необходимо описать, чтобы компилятор мог установить связь между оператором вызова и теми действиями, которые предусмотрены в процедуре (функции). Описание процедуры (функции) помещается в разделе описаний и внешне выглядит как программа, но вместо заголовка программы фигурирует заголовок процедуры (функции).

Не вдаваясь в дальнейшие подробности, попробуем составить собственную процедуру, чтобы пояснить сказанное. Пусть в этой процедуре преобразуется некоторая символьная строка таким образом, чтобы все строчные буквы заменялись соответствующими прописными. В Турбо Паскале имеется стандартная функция UPCASE (см. гл.4), которая выполняет аналогичные действия над одиночным символом. Наша процедура (назовем ее UPSTRING) будет преобразовывать сразу все символы строки,

причем сделаем ее пригодной не только для латинских букв, но и для букв русского алфавита.

Разработку программы проведем в два этапа. Сначала сконструируем основную (вызывающую) часть программы. Ее действия очень просты: она должна ввести входную строку (назовем ее *Sinp*) с клавиатуры, преобразовать ее с помощью процедуры *UpString* в выходную строку *Sout* и напечатать результат. Эти действия нетрудно запрограммировать, на-пример:

```
Program CharsConvert;
```

```
Procedure UpString(s1: String; var s2: String);
```

```
begin {UpString}
```

```
  s2 := s1           {Пока еще нет преобразования!}
```

```
end; {UpString}
```

```
var
```

```
  Sinp, Sout : String; {Исходная и преобразованная строки}
```

```
begin {Начало основной (вызывающей) программы}
```

```
  Write('Введите строку: ');
```

```
  ReadLn(Sinp);           {Вводим исходную строку}
```

```
  UpString(Sinp,Sout); {Преобразуем ее к прописным буквам}
```

```
  WriteLn('      Результат: ',Sout) {Печатаем результат}
```

```
end. {Конец вызывающей программы}
```

В этой программе используется замещение процедуры *UPSTRING* так называемой «заглушкой», т.е. процедурой, в которой на самом деле не осуществляется нужных нам действий, а выходная строка просто копирует входную. (Однако эта программа синтаксически абсолютно правильна и при желании ее можно запустить на счет.) Заглушка понадобилась нам по двум причинам. Во — первых, приведенная программа очень проста, в ней отсутствует детальная реализация процедуры и это позволяет наглядно проиллюстрировать механизм ее описания. Во — вторых, на ее примере мы знакомимся с универсальным методом конструирования сложных программ, получившим название *нисходящее программирование*. В соответствии с этим методом создание программы начинается «сверху», т.е. с разработки самого главного, генерального алгоритма. На верхнем уровне обычно еще не ясны детали реализации той или иной части программы, поэтому эти части следует заменить временными заглушками. Желательно, чтобы временный вариант программы был синтаксически правильным, тогда можно его откомпилировать и убедиться в отсутствии в нем синтаксических ошибок. Такой прогон даст определенную уверенность перед разработкой и реализацией алгоритмов нижнего уровня, т.е. перед заменой заглушек реально работающими процедурами. Если реализуемый в заглушке алгоритм достаточно сложен, его вновь структурируют, выделяя главный алгоритм и применяя новые заглушки, и т.д. Процесс продолжается «вниз» до тех пор, пока не будет создан полностью работоспособный вариант программы.

В дальнейшем мы еще не раз будем использовать метод нисходящего программирования, а сейчас вернемся к описанию нашей процедуры. Как видим, это описание начинается зарезервированным словом **Procedure**, за которым следуют имя процедуры и список формальных параметров. Список параметров заключается в круглые скобки и содержит перечень параметров с указанием их типа. Заметим, что перед параметром *s2*, с помощью которого в вызывающую программу возвращается результат преобразования, стоит зарезервированное слово **VAR**. Именно таким способом компилятору указываются те параметры, в которых процедура возвращает вызвавшей ее программе результат своей работы (подробнее см. гл. 8). Зарезервированное слово **Procedure**, имя процедуры и список ее параметров образуют *заголовок процедуры*. За заголовком следует *тело процедуры*, содержащее новый раздел описаний (этот раздел пока еще пуст) и раздел исполняемых операторов (оператор *s2 := s1*).

Приступим к разработке алгоритма процедуры. Для этого обратимся к таблице кодировки символов, используемой в ПК (см. прил. 2). В соответствии с этой таблицей коды символов латинских строчных букв от *a* до *z* образуют непрерывный массив монотонно нарастающих чисел от 97 до 122, а коды соответствующих им прописных букв — непрерывный массив чисел от 65 до 90. Преобразование строчных латинских букв в прописные, следовательно, состоит в уменьшении кода буквы на 32. Сложнее обстоит дело с символами русского алфавита (кириллицей). В зависимости от принятого способа кодировки русские строчные буквы могут образовывать один сплошной массив (кодировки ГОСТ и МПС), два массива (альтернативная кодировка), несплошной массив (кодировка типа ЕСТЕЛ), неупорядоченный массив (кодировка КОИ—8). Если исключить два последних варианта кодировки, использовавшихся на устаревших ПК, то задача преобразования буквы состоит в том, чтобы к внутреннему коду русской буквы *A* (для букв от *a* до *п*) или к коду буквы *P* (для букв от *p* до *я*) прибавить разницу в кодах текущего символа и кодах букв *a* и *п*. Например, если преобразуется буква *б*, то к коду *A* нужно прибавить разницу между кодами *a* и *б*, т.е. единицу, в результате получим код буквы *Б*. Точно так же при преобразовании буквы *ф* к коду буквы *П* будет прибавлено число 5 (как разница кодов *ф* и *п*), поэтому в результате получится код буквы *Ф*. С учетом этого можно составить следующий алгоритм реализации процедуры: для каждого символа исходной строки *s1* определить, к какому подмассиву *a...z*, *a...p* или *п...я* принадлежит код этого символа, и затем изменить его, добавив к кодам букв *A* (латинская), *A* (русская) или *П* соответствующую разницу. Если символ не принадлежит ни к какому из подмассивов, нужно поместить его код в выходную строку без изменений.

Вот возможный вариант процедуры:

```
Procedure UpString(s1: String; var s2: String);
var
  i: Integer;           {Счетчик цикла преобразования}
  c: Char;             {Рабочая переменная преобразования}
begin                  {UpString}
  s2 := '';             {Вначале выходная строка пуста}
```

```

{Цикл посимвольного анализа исходной строки}
for i := 1 to Length(s1) do
  begin
    {Берем из входной строки очередной символ}
    c := s1[i];
    {Проверяем символ на принадлежность к одному
     из трех подмассивов}
    if (c >= 'a') and (c <= 'z') then
      c := chr(ord('A')+ord(c)-ord('a')) {A,a - латинские!}
    else if (c >= 'а') and (c <= 'п') then
      c := chr(ord('A')+ord(c)-ord('а')) {A,a - русские!}
    else if (c >= 'р') and (c <= 'я') then
      c := chr(ord('P')+ord(c)-ord('р'));
    s2 := s2+c
  end
end; {UpString}

```

В процедуре вначале с помощью оператора

s2 := '';

подготавливается «пустая» выходная строка, т.е. строка нулевой длины. Затем используется цикл от 1 до длины входной строки s1 (эта длина получается с помощью стандартной функции Length), в ходе которого проверяется принадлежность очередного символа указанным подмассивам и осуществляется необходимая коррекция его внутреннего кода. Для доступа к отдельным символам строки используется замечательное свойство типа данных STRING, позволяющее рассматривать строку как набор (массив) символов. Первый символ этого набора имеет индекс 1, второй — 2 и т.д. Индекс указывается сразу за именем строки в квадратных скобках. Таким образом, s1[i] — это i-ый символ строки s1. Преобразованный символ добавляется в конец выходной строки.

Добавив комментарии и поместив тело процедуры вместо заглушки в первоначальный вариант программы, получим окончательно ее рабочий вариант (пример 2.10).

Пример 2.10

Program CharsConvert;

{Программа вводит произвольную текстовую строку, преобразует все входящие в нее буквы в прописные и печатает результат преобразования}

PROCEDURE UpString(s1 : String; var s2 : String);

{Эта процедура преобразует буквы входной строки s1 в прописные буквы латинского или русского алфавита и помещает результат преобразования в выходную строку s2. Используется предположение о том, что последовательности латинских букв от «а» до «z» и русских букв от «а» до «п» и от «р» до «я», а также по-

следовательности соответствующих им прописных букв образуют непрерывные массивы}

```

var
  i: Integer;           {Счетчик цикла преобразования}
  c: Char;              {Рабочая переменная преобразования}
begin
  {UpString}
  s2 := '';             {Вначале выходная строка пуста}
  {Цикл посимвольного анализа исходной строки}
  for i := 1 to Length(s1) do
    begin
      {Берем из входной строки очередной символ}
      c := s1[i];
      {Проверяем символ на принадлежность к одному
       из трех подмассивов}
      if (c >= 'a') and (c <= 'z') then
        c := chr(ord('A')+ord(c)-ord('a')) {A,a - латинские!}
      else if (c >= 'а') and (c <= 'п') then
        c := chr(ord('A')+ord(c)-ord('а')) {A,a - русские!}
      else if (c >= 'р') and (c <= 'я') then
        c := chr(ord('P')+ord(c)-ord('р'));
      s2 := s2+c
    end
  end;
  {UpString}

var
  Sinp, Sout : String; {Исходная и преобразованная строки}
begin {Начало основной (вызывающей) программы}
  Write('Введите строку: ');
  ReadLn(Sinp);           {Вводим исходную строку}
  UpString(Sinp, Sout);    {Преобразуем ее к прописным буквам}
  WriteLn('    Результат: ', Sout) {Печатаем результат}
end. {Конец вызывающей программы}

```

Рассмотрим иной способ реализации той же программы: оформим алгоритм преобразования в виде функции. Кроме того, с помощью стандартной функции `UPCASE` преобразуем каждый очередной символ (это преобразование осуществляется только для букв латинского алфавита) и тем самым исключим проверку принадлежности символа к строчным латинским буквам:

```

Function UpString(s1: String): String;
var
  i : Integer;
  c : Char;
  s2: String;           {Результат преобразования}
begin {UpString}
  s2 := '';
  for i := 1 to Length(s1) do

```



```
begin
  {Получаем и преобразуем очередной символ}
  c := UpCase(sl[i]);
  if (c >= 'a') and (c <= 'п') then
    c := chr(ord('A')+ord(c)-ord('a'))
  else
    if (c >= 'p') and (c <= 'я') then
      c := chr(ord('P')+ord(c)-ord('p'));
    s2 := s2+c
  end;
  UpString := s2      {Присваиваем значение функции UpString}
end; {UpString}

var
  Sinp: String;
begin {Начало основной программы}
  Write('Введите строку: ');
  ReadLn(Sinp);
  WriteLn('      Результат: ', UpString(Sinp))
end. {Конец основной программы}
```

Программа получилась несколько проще за счет того, что функцию можно использовать в качестве параметра обращения к другой процедуре (в нашем случае к WriteLn). Обратите внимание: в теле любой функции нужно осуществить присваивание ей вычисленного значения (см. оператор UpString := s2). В левой части оператора присваивания в этом случае указывается имя функции.

2.7. ПРИМЕРЫ ПРОГРАММ

Мы познакомились с основными возможностями языка Турбо Паскаль. Как видите, ядро языка очень компактно и отличается простотой — именно в этом состоит главная заслуга автора Паскаля Н.Вирта: язык, придуманный им, прост и естественен, он легко осваивается, на нем не трудно писать самые разнообразные программы. Конечно, рассмотрены далеко не все свойства Турбо Паскаля, ведь его главная отличительная черта — это богатство типов данных. Однако уже рассмотренного вполне достаточно для написания многих полезных программ.

Приводимые ниже программы относительно сложны, поэтому они реализуются поэтапно, по методу нисходящего программирования. Мне кажется, что тем читателям, кто не имеет большого опыта в программировании или кто захочет подробнее ознакомиться с нисходящим программированием, изучение этой главы принесет определенную пользу. Если Вам будет скучно разбираться в «кухне» программирования, но Вас заинтересуют описываемые здесь программы и Вы захотите их повторить, то в прил.5 Вы найдете полный текст соответствующей программы; однако в каждой из них используются некоторые дополнительные возможности языка Турбо Паскаль, которые не рассматривались ранее и которые сб-суждаются в пропущенных Вами фрагментах книги.

При оформлении программ я стремился использовать *хороший стиль* написания программ, т.е. такую их форму, которая дает наиболее полное представление о структуре программы в целом и ее отдельных частей. Не существует какого-либо стандарта, определяющего хороший стиль программы. Обычно это интуитивное понятие включает способ расположения операторов и описаний по строкам (не рекомендуется размещать более одного оператора на каждой строке), а также выделение отступами тела составных и условных операторов. Последнее особенно важно в программах Турбо Паскаля: сплошь и рядом в них встречаются операторные скобки **begin... end**, причем часто вложенные друг в друга; использование отступа служит дополнительным средством проверки правильности их расстановки — не случайно в редакторе среды предусмотрена соответствующая опция. Принятый мною стиль оформления программ не претендует на эталон, просто мне кажется, что таким образом оформленные программы читаются лучше. Если Вы всерьез намерены программировать на Турбо Паскале, имеет смысл составить собственное представление о хорошем стиле и далее неукоснительно придерживаться его — очень скоро некоторые дополнительные издержки на подготовку программ с лихвой окупятся их «читабельностью», а это поможет Вам вспомнить все детали реализации программы, которая была написана несколько месяцев тому назад.

2.7.1. Вычисление дня недели

Случалось ли Вам мучительно вспоминать, какой именно день недели приходился на то или иное число год или два назад, или вычислять, на какой день недели в этом году приходится Ваш день рождения? Если да, то Вас, думаю, заинтересует простая программа, позволяющая по заданной дате мгновенно вычислить соответствующий день недели. В ее основе лежит такая формула:

день недели = остаток от деления X на 7,

где $X = \text{abs}(\text{trunc}(2.6 * m - 0.2) + d + y / 4 + y * c / 4 - 2 * c)$;

m — номер месяца (см. ниже);

d — число (день месяца);

c — номер столетия (см. ниже);

y — номер года в столетии.

При использовании этой формулы следует учесть два обстоятельства. Во-первых, формула верна для григорианского календаря нового стиля (от 1582 до 4903 года). Во-вторых, год и месяц следует предварительно преобразовать так, как если бы начало года приходилось на 1 марта. Иными словами, март в этой формуле имеет порядковый номер 1, апрель 2, ... , январь 11 и февраль 12, причем январь и февраль следует отнести к предыдущему году. Например, для 1 февраля 1991 года номер месяца должен быть равен 12, а год 1990, в то время как для 31 декабря 1991 года номер месяца — 10, а год — 1991. Результат вычисления дается в виде целого числа в диапазоне от 0 до 6, причем 0 соответствует воскресенью.

Приступим к разработке программы. Прежде всего, предположим, что программа уже создана и Вы осуществляете ее прогон. Какая форма взаимодействия с программой кажется Вам наиболее подходящей? Вряд ли Вас удовлетворит однократное ее исполнение (ввод некоторой даты и вывод на экран соответствующего дня недели). Скорее всего Вы захотите повторить работу программы для нескольких дат, например, поинтересоваться, в какой день недели Вы родились, затем, на какой день недели приходится в этом году Ваш день рождения, дни рождения близких, друзей; может быть, определить, в какой день родились известные Вам исторические деятели, и т.д. Таким образом, в программе следует предусмотреть многократное выполнение действий *<ввод даты>* — *<вычисление дня недели>*, причем число циклов вычисления заранее не известно. Сразу же возникает новый вопрос: как сообщить программе, что Вы завершаете работу с ней? Для этого можно условиться, что ввод некоторой заранее обусловленной или недопустимой даты должен интерпретироваться программой, как указание на прекращение работы. С учетом сказанного, напомним такой начальный вариант программы:

```
var
  IsCorrectDate: Boolean; {Признак правильной даты}
  d,m,y       : Integer; {Вводимая дата - день, месяц и год}
begin
  repeat
    {Ввести в переменные d, m и y очередную дату и
     проверить ее. Если дата правильная, установить
     IsCorrectDate=True, иначе IsCorrectDate=False}
    if IsCorrectDate then
      {Вычислить и выдать на экран день недели};
  until not IsCorrectDate
end.
```

Если Вы попытаетесь запустить эту программу на счет, то ее поведение будет зависеть от начального значения переменной IsCorrectDate. Это значение случайно, так как компилятор Турбо Паскаля не проводит начальной инициализации переменных. Скорее всего, тот байт оперативной памяти, в котором она разместится, окажется нулевым, что в Турбо Паскале расценивается как логическое значение FALSE, поэтому с большой вероятностью ничего не произойдет, и программа сразу же завершит свою работу (условие **not** IsCorrectDate будет выполнено). Если начальное значение IsCorrectDate окажется не нулевым, то цикл **REPEAT...UNTIL** будет выполняться до тех пор, пока Вы не выключите компьютер или не нажмете клавиши *Ctrl-Break*.

Будем считать, что необходимые действия осуществляются в двух процедурах с именами InputDate (ввод даты) и WriteDay (вычисление и печать дня недели). В процедуру InputDate не нужно ничего передавать из программы, так как в ней самой осуществляются ввод и контроль даты. Поэтому заголовок процедуры может иметь такой вид:

```
Procedure InputDate(var d,m,y: Integer; var correctly:
                    Boolean);
```

Процедура WriteDay, напротив, только получает из программы нужные ей данные и ничего не возвращает в программу, поэтому в ее заголовке параметры описываются без слова **VAR**:

```
Procedure WriteDay(d,m,y : Integer);
```

С учетом этого программу можно уточнить следующим образом:

```
var
  IsCorrectDate: Boolean; {Признак правильной даты}
  d,m,y        : Integer; {Вводимая дата - день, месяц и год}
{-----}
Procedure InputDate(var d,m,y : Integer;
                    var correctly : Boolean);
{Вводит в переменные d, m и y очередную дату и проверяет ее.
Если дата правильная, устанавливает correctly=true, иначе
correctly=false }
begin {InputDate}
  correctly := false
end; {InputDate}
{-----}
Procedure WriteDay(d,m,y: Integer);
{Вычисляет день недели и выводит его на экран}
begin {WriteDay}
end; {WriteDay}
{-----}
begin
  repeat
    InputDate(d,m,y, IsCorrectDate);
    if IsCorrectDate then
      WriteDay(d,m,y)
  until not IsCorrectDate
end.
```

Теперь можно разработать процедуру INPUTDATE. Ввод даты не вызывает трудностей — стандартные процедуры WRITE и READLN отлично приспособлены для этой цели. Для проверки правильности даты нужно проверить принадлежность месяца диапазону 1...12 и года — диапазону 1582...4903. Кроме того, число не должно выходить из диапазона 1...31. Если Вы не очень настаиваете на более точной проверке числа в зависимости от месяца и года (для февраля), то программная реализация процедуры будет следующей:

```
Procedure InputDate(var d,m,y : Integer;
                    var correctly : Boolean);
{Вводит в переменные d, m и y очередную дату и проверяет ее.
Если дата правильная, устанавливает correctly=true, иначе
```

```

correctly=false }
begin {InputDate}
  Write('Введите дату в формате ДД ММ ГГГГ: ');
  ReadLn(d,m,y);
  correctly := (d>=1) and (d<=31) and (m>=1)
               and (m<=12) and (y>=1582) and (y<=4903)
end; {InputDate}

```

При выполнении этой процедуры ввод, например, трех нулей приведет к присвоению переменной CORRECTLY значения FALSE, что вызовет завершение работы программы.

Теперь разберемся с процедурой WRITEDAY. Получив в параметрах обращения день, месяц и год, она должна:

- преобразовать месяц и год так, как описано выше (год должен начинаться 1 марта);
- вычислить день недели;
- выдать на экран результат.

Первое и второе действия очень просты и легко программируются. Что касается выдачи на экран, то можно потребовать от программы, чтобы эта выдача была не просто числом от 0 до 6, а одной из строк «воскресенье», «понедельник», ..., «суббота». Для этого потребуются дополнительные усилия: нужно сначала создать массив строковых констант с именем, например, DAYS_OF_WEEK (*дни недели*), а затем выбрать из этого массива и выдать на экран нужную строку. Создать массив текстовых констант можно с помощью объявления типизированной константы (см. гл. 7):

const

```

Days_of_week: array [0..6] of String [11] =
  ('воскресенье', 'понедельник', 'вторник',
   'среда', 'четверг', 'пятница', 'суббота');

```

В этом объявлении идентификатор Days_of_week описывается в разделе констант, однако справа от него указан тип данных (массив строк), как если бы описывалась переменная, а уже только после типа стоит знак равенства и заключенный в круглые скобки список элементов массива. В результате получим следующую процедуру:

Procedure WriteDay(d,m,y : Integer);

const

```

Days_of_week: array [0..6] of String [11] =
  ('воскресенье', 'понедельник', 'вторник',
   'среда', 'четверг', 'пятница', 'суббота');

```

var

c, w : Integer;

begin

if m < 3 then

begin

m := m + 10;

y := y - 1

{Месяц январь или февраль}

```

    end
else
    m := m - 2;           {Остальные месяцы}
    c := y div 100;       {Вычисляем столетие}
    y := y mod 100;       {Находим год в столетии}
    w := abs(trunc(2.6*m-0.2)+d+y div 4+y+c div 4-2*c) mod 7;
    WriteLn(Days_of_week[w])
end;

```

Окончательный вариант программы приведен в прил.5.1.

2.7.2. Биоритмы

Давно известно, что творческая и физическая активность человека не остается постоянной, циклически меняется, причем периодичность ее изменения приблизительно согласуется с периодом вращения Луны вокруг Земли. Существует теория, согласно которой физическая, эмоциональная и интеллектуальная активность человека подчиняется соответствующим биоритмам. Каждый биоритм представляет собой синусоиду со строго постоянным периодом, причем для каждого биоритма существует свой период. В отдельные дни все три биоритма человека могут достигнуть своего максимума и тогда человек испытывает подъем творческих и физических сил, в такие дни у него все спорится, он легко решает проблемы, которые в другое время ему решить гораздо сложнее. Точно также существуют и «черные» дни, соответствующие спаду всех трех биоритмов.

Используя уже опробованную методику нисходящего программирования, создадим программу, в которой запрашивается дата рождения человека и дата, для которой требуется оценить его состояние. Программа должна рассчитать и выдать на экран ближайшие к этой дате дни пика и спада биоритмов.

Алгоритм программы можно укрупненно записать следующим образом:

- ввести дату рождения и текущую дату, проконтролировать их правильность и непротиворечивость;
- вычислить количество дней между двумя датами, чтобы определить фазу синусоид для текущей даты;
- вычислить количество дней от текущей даты до даты ближайшего пика биоритмов и даты ближайшего спада;
- определить и напечатать обе даты.

Будем считать, что каждое из перечисленных действий реализуется в отдельной процедуре, тогда начальный вариант программы будет таким:

```

Procedure InputDates(var d0,m0,y0,d,m,y: Integer);
  {Вводит дату рождения и текущую дату. Контролирует правильность дат и их непротиворечивость (текущая дата должна быть позже даты рождения) }
begin {InputDates}
end; {InputDates}

{-----}
Procedure Get_count_of_days(d0,m0,y0,d,m,y: Integer);

```

```

                                var days: Integer);
{Определяет полное количество дней, прошедших от одной даты
 до другой}
begin {Get_count_of_days}
end; {Get_count_of_days}
{-----}
Procedure FindMaxMin(var dmin,dmax: Integer; days: Integer);
{Ищет критические дни}
begin {FindMaxMin}
end; {FindMaxMin}
{-----}
Procedure WriteDates(dmin,dmax,days: Integer);
{Определяет критические даты по количеству дней, прошедших от
 момента рождения, и выдает эти даты на экран}
begin {WriteDates}
end; {WriteDates}
{-----}
var
    d0,d,           {Дни рождения и текущий}
    m0,m,           {Месяцы рождения и текущий}
    y0,y,           {Годы рождения и текущий}
    dmin,           {Наименее благоприятный день}
    dmax,           {Наиболее благоприятный день}
    days: Integer; {Количество дней от рождения}
begin {Главная программа}
    InputDates(d0,m0,y0,d,m,y);
    Get_numbers_of_days(d0,m0,y0,d,m,y,days);
    FindMaxMin(dmin,dmax,days);
    WriteDates(dmin,dmax,days)
end.

```

Начинаем детализацию программы. Прежде всего подумаем, как по двум датам вычислить разделяющее их количество дней? Если вспомнить, что следует учитывать неодинаковое количество дней по месяцам года, а также 29 февраля для високосных лет, то ответ на этот вопрос окажется не таким уж простым. Предлагаемый алгоритм подсчета количества дней заключается в вычислении количества дней от даты рождения до конца месяца, а затем и года рождения, количества дней от начала текущего года до текущего месяца и текущей даты, а также — в подсчете количества полных лет, разделяющих обе даты. Количество лет затем легко пере- считывается в количество дней с учетом длины года (365 дней для обычных и 366 дней для високосных лет). Это очень прямолинейный алгоритм, но, откровенно говоря, мне не пришло в голову ничего другого. Возможно, существует более изящный способ подсчета и Вы его знаете, тогда про- граммная реализация будет другой.

Упростить алгоритм можно за счет создания и использования массива из 12 целых чисел, содержащего количества дней по месяцам невисо- косного года, т.е. 31, 28, 31, 30 и т.д. Этот массив (назовем его

SIZE_OF_MONTH — *длина_месяца*) можно использовать и для обратной задачи, т.е. для определения даты критических дней, а также для проверки правильности вводимых дат. Таким образом, массив SIZE_OF_MONTH будет использоваться сразу в трех процедурах. Сделаем его глобальным, для чего его описание поместим перед описанием процедур:

```
const
  Size_of_Month: array [1..12] of Byte =
    (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
{-----}
Procedure InputDates(var d0,m0,y0,d,m,y: Integer);
.....
```

Поскольку описание массива размещается до описания процедур, он становится доступным внутри каждой из процедур и служит для них глобальным. В отличие от этого все константы и переменные, объявляемые внутри некоторой процедуры, являются локальными и могут использоваться только в этой процедуре.

С учетом сказанного напомним следующий начальный вариант программной реализации процедуры INPUTDATES:

```
Procedure InputDates(var d0,m0,y0,d,m,y: Integer);
{Вводит дату рождения и текущую дату. Контролирует правильность дат и их непротиворечивость (текущая дата должна быть позже даты рождения)}
var
  correctly: Boolean; {Признак правильного ввода}
begin {InputDates}
  repeat
    {Вводим и контролируем дату рождения d0,m0,y0.}
    {Вводим и контролируем текущую дату d,m,y.}
    {Проверяем непротиворечивость дат:}
    correctly := y > y0;
    if not correctly and (y = y0) then
      begin
        correctly := m > m0;
        if not correctly and (m = m0) then
          correctly := d >= d0
        end
      until correctly
  end; {InputDates}
```

В этой процедуре дважды выполняется одно и то же алгоритмическое действие (ввод и контроль даты). Это действие можно вынести в отдельную внутреннюю процедуру с именем INPDATE, тогда получим следующий окончательный вариант:

```
Procedure InputDates(var d0,m0,y0,d,m,y : Integer);
```


{Вводит дату рождения и текущую дату. Контролирует правильность дат и их непротиворечивость (текущая дата должна быть позже даты рождения)}

var

correctly: Boolean; {Признак правильного ввода}
{-----}

Procedure InpDate(text: **String**; var d,m,y: Integer);

{Выводит приглашение TEXT, вводит дату в формате ДД ММ ГГГГ и проверяет ее правильность}

const

YMIN = 1800; {Минимальный правильный год}
YMAX = 2000; {Максимальный правильный год}

begin {InpDate}

repeat

Write(text);

ReadLn(d,m,y);

correctly := (y >= YMIN) and (Y <= YMAX) and (m >= 1)
and (m <= 12) and (d > 0);

if correctly **then**

if (m = 2) and (d = 29) and (y mod 4 = 0)

then

{Ничего не делать: это 29 февраля високосного года!}

else

correctly := d <= Size_of_Month[m];

if not correctly **then**

WriteLn('Ошибка в дате!')

until correctly

end; {InpDate}

{-----}

begin {InputDates}

repeat

InpDate(' Введите дату рождения в формате ДД ММ ГГГГ:',
d0,m0,y0);

InpDate(' Введите текущую дату: ',d,m,y);

{Проверяем непротиворечивость дат:}

correctly := y > y0;

if not correctly and (y = y0) **then**

begin

correctly := m > m0;

if not correctly and (m = m0) **then**

correctly := d >= d0

end

until correctly

end; {InputDates}

В самом общем виде алгоритм подсчета количества дней, разделяющих две даты, описан выше. При его реализации следует учесть три возможных варианта:

- месячный младенец (год и месяц обеих дат одинаков): количество дней находится простым вычитанием двух чисел;
- годовалый младенец (год обеих дат совпадает): количество дней = (остаток дней в месяце рождения) + (количество дней в текущем месяце) + (количество дней в месяцах, разделяющих обе даты);
- общий вариант (отличаются года): количество дней = (количество дней от даты рождения до конца года) + (количество дней в разделяющих даты годах) + (количество дней от начала текущего года до текущей даты).

С учетом этого составим начальный вариант программной реализации процедуры GET_NUMBERS_OF_DAYS:

```

Procedure Get_numbers_of_days(d0,m0,y0,d,m,y: Integer;
                               var days: Integer);
{Определение полного количества дней, прошедших от одной даты
 до другой }
{-----}
Procedure Variant2;
{Подсчет количества дней в месяцах, разделяющих обе даты}
begin {Variant2}
end; {Variant2}
{-----}
Procedure Variant3;
{Подсчет количества дней в месяцах и годах, разделяющих обе
 даты}
begin {Variant3}
end; {Variant3}
{-----}
begin {Get_numbers_of_days}
  if (y = y0) and (m = m0) then {Даты отличаются только днями:}
    days := d - d0
  else {Даты отличаются не только днями:}
    begin
      days := d + Size_of_Month[m0] - d0;
      {Учитываем количество дней в текущем месяце и
       количество дней до конца месяца рождения}
      if (y0 mod 4 = 0) and (m0 = 2) then
        inc(days); {Учитываем високосный год}
      if y = y0 then
        Variant2 {Разница в месяцах одного и того же года!}
      else
        Variant3 {Даты отличаются годами}
      end
    end; {Get_numbers_of_days}

```

В этом фрагменте используется способ связи вспомогательных процедур VARIANT2 и VARIANT3 с основной процедурой через глобальные переменные, которыми являются параметры обращения к основной процедуре.

Вспомогательные процедуры удобнее всего реализовать на основе циклов **WHILE**:

Procedure Variant2;

{Подсчет количества дней в месяцах, разделяющих обе даты }

var

mm : Integer;

begin {Variant2}

mm := m0;

while mm < m **do**

begin

days := days + Size_of_Month[mm];

if (mm = 2) **and** (y0 mod 4 = 0) **then**

inc(days);

inc(mm)

end

end; {Variant2}

{-----}

Procedure Variant3;

{Подсчет количества дней в месяцах и годах, разделяющих обе даты }

var

mm, yy : Integer;

begin {Variant3}

mm := m0 + 1;

while mm <= 12 **do** *{Учитываем остаток года рождения:}*

begin

days := days + Size_of_Month[mm];

if (mm = 2) **and** (y0 mod 4 = 0) **then**

inc(days);

inc(mm)

end;

yy := y0 + 1;

while yy < y **do** *{Прибавляем разницу лет:}*

begin

days := days + 365;

if yy mod 4 = 0 **then**

inc(days);

inc(yy)

end;

mm := 1;

while mm < m **do** *{Прибавляем начало текущего года:}*

begin

days := days + Size_of_Month[mm];

if (y mod 4 = 0) **and** (mm = 2) **then**

inc(days);

inc(mm)

end

end; {Variant3}

В процедуре FINDMAXMIN осуществляется поиск критических дней, т.е. ближайших к текущей дате дней, для которых все три биоритма достигают своего максимума и минимума. Предполагается, что биоритмы изменяются по законам синуса от количества прожитых дней с периодами T_F , T_E и T_I соответственно для физической, эмоциональной и интеллектуальной активности человека. В программе приняты следующие периоды (в днях):

$$T_F = 23.6884$$

$$T_E = 28.4261$$

$$T_I = 33.1638$$

Самый простой алгоритм поиска заключается в том, чтобы вычислить значения сумм всех трех синусов для текущего дня и для каждого из последующих дней на некотором заранее обусловленном интервале, на — пример, в пределах месяца. Сопоставив результаты расчетов для каждого дня, нетрудно определить критические дни:

```
Procedure FindMaxMin(var dmin,dmax: Integer;
                      days: Integer);
{Поиск критических дней}
const
  TF = 2*3.1416/23.6884; {Период физической активности}
  TE = 2*3.1416/28.4261; {Период эмоциональной активности}
  TI = 2*3.1416/33.1638; {Период интеллектуальной активности}
  INTERVAL = 30;         {Интервал прогноза}
var
  min,                    {Накапливает минимум биоритмов}
  max,                    {Накапливает максимум биоритмов}
  x : real;               {Текущее значение биоритмов}
  i : Integer;
begin {FindMaxMin}
  max := sin(days*TF)+sin(days*TE)+sin(days*TI);
  min := max; {Начальное значение минимума и максимума равно
               значению биоритмов для текущего дня}

  dmin := days;
  dmax := days;
  for i := 0 to INTERVAL do
    begin
      x := sin((days+i)*TF) + sin((days+i)*TE) +
           sin((days+i)*TI);
      if x > max then
        begin
          max := x;
          dmax := days + i
        end
      else if x < min then
        begin
```

```

    min := x;
    dmin := days + i
  end
end;
end; {FindMaxMin}

```

При разработке алгоритма процедуры WRITEDATES, с помощью которой на экран выводится результат работы программы, учтем, что основные сложности будут связаны с определением новой даты по начальной дате и количеству прошедших дней. Этот расчет будет повторяться дважды — для даты пика и даты спада биоритмов, поэтому его следует вынести в отдельную процедуру WRITEDATES. Кроме того, вряд ли Вы откажетесь от возможности вывода на экран дополнительной информации о том, сколько полных дней, часов, минут и секунд разделяют дату рождения человека и текущую дату. Однако реализация этого вывода не столь проста, как это может показаться на первый взгляд. Дело в том, что диапазон возможных значений данных типа INTEGER составляет от -32768 до $+32767$. Средняя продолжительность жизни человека — около 70 лет, т.е. 25550 дней. Это значение еще можно представить в переменной типа INTEGER, однако часы, минуты и тем более секунды средней продолжительности жизни далеко превышают этот диапазон. Чтобы получить вывод достоверных данных, необходимо расширить диапазон значений целых чисел. Для этого в Турбо Паскале предусмотрен специальный тип данных LONGINT («длинный» целый), имеющий диапазон значений от -2147483648 до $+2147483647$ (см. гл. 4). Поэтому в процедуре WRITEDATES следует предусмотреть вспомогательную переменную этого типа, присвоить ей значение переменной DAYS и уже затем использовать «длинную» переменную для вычисления (и вывода) часов, минут, секунд. В результате начальный вариант процедуры WRITEDATES может быть таким:

```

Procedure WriteDates(dmin,dmax,days : Integer);

```

{Определение и вывод дат критических дней.

Вывод дополнительной информации о количестве
прожитых дней, часов, минут и секунд }

```

{-----}

```

```

Procedure WriteDate(text : String; dd : Integer);

```

{Определение даты для дня DD от момента рождения. В глобальных переменных d, m и y имеется текущая дата, в переменной DAYS — количество дней, прошедших от момента рождения до текущей даты.

Выводится сообщение TEXT и найденная дата в формате
ДД-МЕС-ГГГГ}

```

begin {WriteDate}

```

```

end; {WriteDate}

```

```

{-----}

```

```

var

```

LongDays: LongInt; {"Длинная" целая переменная для часов,
минут и секунд }

```

begin {WriteDates}

```

```

LongDays := days;
WriteLn('Прошло: ', LongDays, ' дней, ', longDays*24,
        ' часов, ', LongDays*24*60, ' минут, ',
        LongDays*24*60*60, ' секунд');
WriteDate('Наименее благоприятный день: ', dmin);
WriteDate('Наиболее благоприятный день: ', dmax)
end; {WriteDates}

```

Реализация процедуры WRITEDATE не вызывает особых сложностей:

```

Procedure WriteDate(text: String; dd: Integer);
const
    Names_of_Monthes : array [1..12] of String [3] =
        ('январь', 'фев', 'мар', 'апр', 'мая', 'июн',
         'июл', 'авг', 'сен', 'окт', 'ноя', 'дек');
var
    d0, m0, y0, ddd : Integer;
begin {WriteDate}
    d0 := d;
    m0 := m;
    y0 := y;
    ddd := days;
    while ddd <> dd do
        begin
            inc(d0); {Наращиваем число}
            if (y0 mod 4 <> 0) and (d0 > Size_of_Month[m0]) or
                (y0 mod 4=0) and (d0=30) then
                begin {Корректируем месяц}
                    d0 := 1;
                    inc(m0);
                    if m0 = 13 then {Корректируем год}
                        begin
                            m0 := 1;
                            inc(y0)
                        end
                    end;
                    inc(ddd)
                end;
            end;
        end;
    WriteLn(text, d0, '-', Names_of_Monthes[m0], '-', y0)
end; {WriteDate}

```

Собрав воедино отдельные части, получим полный текст программы (прил.5.2), предназначенной для определения биоритмов.

2.7.3. Игра ним

Ним — одна из самых старых и увлекательных математических игр. Для игры в ним необходим партнер (в ним играют вдвоем), стол и набор фишек. В качестве фишек обычно используются камешки или монетки. В

наиболее известном варианте нима 12 фишек раскладываются в три ряда так, как показано на рис. 2.3.

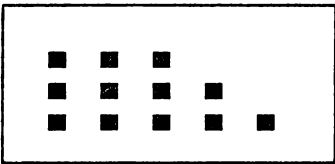


Рис.2.3. Фишки, расположенные для игры в ним по схеме 3–4–5

Правила нима просты. Игроки по очереди забирают одну или несколько фишек из любого ряда. Не разрешается за один ход брать фишки из нескольких рядов. Выигрывает тот, кто возьмет последнюю фишку (фишки).

Если Вы сыграете несколько партий в ним, то скоро заметите, что существует некоторая оптимальная последовательность ходов, которая гарантирует победу, если только Вы начинаете игру и первым ходом берете две фишки из первого ряда. Любой другой ход даст шанс Вашему сопернику, который в этом случае наверняка победит, если, в свою очередь, воспользуется оптимальной стратегией.

Полный анализ игры с обобщением на любое число рядов с любым числом фишек в каждом ряду впервые опубликовал в 1901 г. профессор математики из Гарвардского университета Чарльз Л.Бутон, который и назвал игру «ним» от устаревшей формы английских глаголов «стянуть», «украсть». Открытая им оптимальная стратегия основана на двоичной системе счисления и довольно проста. Каждую комбинацию фишек Бутон назвал либо опасной, либо безопасной: если позиция, создавшаяся после очередного хода игрока, гарантирует ему победу, она называется безопасной, если такой гарантии нет — опасной. Бутон строго доказал, что любую опасную позицию всегда можно превратить в безопасную нужным ходом. Наоборот, если перед очередным ходом игрока уже сложилась безопасная позиция, то любой его ход превращает позицию в опасную. Таким образом, оптимальная стратегия состоит в том, чтобы каждым ходом опасную позицию превращать в безопасную и заставлять противника «портить» ее. Использование оптимальной стратегии гарантирует победу игроку только тогда, когда он открывает партию и начальная позиция фишек опасна или он делает второй ход, а начальная позиция безопасна.

Чтобы определить, опасна позиция или безопасна, нужно количество фишек в каждом ряду записать в двоичной системе счисления. Если сумма чисел в каждом столбце (разряде) равна нулю или четна, позиция безопасна. Если же сумма нечетна хотя бы в одном разряде, то позиция опасна. Например, для начальной позиции по схеме 3–4–5 получим:

Десятичная запись количества фишек	Двоичная запись количества фишек
3	011
4	100
5	101
Сумма по разрядам	212

Сумма цифр в среднем столбце равна 1 — нечетному числу, что свидетельствует об опасности этой позиции. Поэтому первый игрок может сделать ее безопасной для себя, если он возьмет две фишки из первого

ряда. В результате в первом ряду остается только 1 фишка (двоичное число также 1), и сумма чисел в среднем столбце изменится на ноль.

В привычной нам десятичной системе счисления емкость каждого разряда равна 10, а для записи значений разряда используются цифры от 0 до 9. В двоичной системе счисления емкость каждого разряда равна 2, а из всех цифр используются только 0 и 1. В этой системе число записывается в виде суммы степеней двойки и при переходе от одного разряда к соседнему левому вес разряда увеличивается в 2 раза. Если нужно записать число 2 в двоичной системе, следует действовать точно так же, как при записи числа 10 в десятичной системе: записать ноль в первом (младшем) разряде и единицу — слева от него, т.е. 10 в двоичной системе означает 2 в десятичной системе. Точно так же 100 в двоичной системе означает 4 в десятичной, 1000 — 8 и т.д.

Для перевода любого целого положительного числа из десятичной системы в двоичную можно использовать прием последовательного деления числа на 2. Например, для перевода десятичного числа 11 в двоичную систему используется такая цепочка делений:

Делимое	Результат	Остаток
11	5	1
5	2	1
2	1	0

Если, начиная с последнего результата, остатки от деления записать в обратном порядке, получим 1011 — это и есть двоичное представление десятичного числа 11. В этом легко

убедиться, записав двоичное число 1011 как сумму степеней 2:

$$1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 = 11$$

Попробуем разработать программу, которая будет выполнять роль партнера в игре, причем это будет весьма опасный противник, так как он будет «знать» оптимальную стратегию и умело ею пользоваться.

Представим себе на минутку, что Вы уже создали программу и начинаете работу с ней. Как организовать удобное взаимодействие с программой? Конечно, возможно простейшее решение: Вы предварительно раскладываете на столе монетки, по запросу программы вводите в нее Ваш ход, затем читаете на экране ход программы, делаете нужные изменения в раскладке монет и т.д. Вряд ли Вас удовлетворит такая программа. Гораздо эффективнее имитировать на экране игровое поле с фишками и своеобразием табло игры, в котором сообщается об очередных ходах противников. Однако использованные ранее средства вывода данных (процедуры WRITE и WRITELN) недостаточны для этих целей, ведь с их помощью нельзя указать конкретное место на экране, куда нужно поместить выводимую информацию. Вывод этими процедурами всегда начинается с той позиции на экране, которую в данный момент занимает курсор. Следовательно, для вывода текста в нужное место экрана требуется перед обращением к этим процедурам изменить положение курсора. Для этих целей служит процедура GOTOXY, которая хотя и является стандартной, но располагается в отдельной библиотеке (модуле) с именем CRT. Подробнее о модулях мы поговорим в гл.9, а сейчас просто учтем, что процедуры и функции из дополнительных библиотек становятся доступны в программе, если в самом ее начале объявить об их использовании. Так, указание об использовании библиотеки CRT делается таким образом:

Uses CRT;

После такого указания программе становятся доступны дополнительные процедуры и функции, с помощью которых можно организовать гибкое управление текстовым экраном, в том числе процедура GOTOXY, перемещающая курсор в произвольное место на экране.

Теперь попробуем составить алгоритм главной программы. В простейшем виде он таков:

```
Uses CRT; {Подключение библиотеки дополнительных
            процедур и функций для управления экраном}
var
    exit: Boolean; {Признак окончания работы}
begin
    {Подготовить экран к работе}
    repeat
        {Ввести, проконтролировать и отобразить ход игрока}
        {Найти и отобразить ход программы}
    until exit
end.
```

В этом алгоритме выделяются три главных действия и организуется цикл, который будет выполняться до тех пор, пока где-то в программе переменной EXIT (выход) не будет присвоено значение TRUE.

Вначале экран подготавливается к работе: формируется игровое поле с фишками и выводится информация о правилах игры. Как уже говорилось, ним позволяет играть с произвольным количеством фишек. Разумно ввести в программу возможность, которая бы позволила пользователю самому указывать число рядов и количество фишек в рядах, т.е. настраивать программу на нужную раскладку фишек. Можно несколько модифицировать главную программу, чтобы предусмотреть эту возможность:

```
Uses CRT; {Подключение библиотеки дополнительных
            процедур и функций для управления экраном}
var
    exit    : Boolean; {Признак окончания работы}
    change  : Boolean; {Признак изменения условий игры}
    {-----}
Procedure Prepare;
    {Готовит экран к игре}
begin {Prepare}
end;  {Prepare}
    {-----}
Procedure GetPlayerMove;
    {Получает, контролирует и отображает ход игрока}
begin {GetPlayerMove}
end;  {GetPlayerMove}
    {-----}
Procedure SetOwnerMove;
```

```

{Находит и отображает очередной ход программы}
begin {SetOwnerMove}
end; {SetOwnerMove}
{-----}
begin {Главная программа}
  {Подготовить начальную расстановку фишек}
  repeat                                     {Цикл изменения условий игры}
    Prepare;                               {Подготовить экран}
    repeat                                   {Игровой цикл}
      GetPlayerMove;                       {Получить ход пользователя}
      if not (exit or change) then
        SetOwnerMove                       {Определить собственный ход}
      until exit or change
    until exit
  end.

```

В этом варианте главная программа содержит два вложенных друг в друга цикла **Repeat...Until**: внутренний цикл управляет игрой, внешний отвечает за изменение условий игры. Оба цикла управляются двумя логическими переменными, которые являются глобальными для трех основных процедур **PREPARE**, **GETPLAYERMOVE**, **SETOWNERMOVE** и, следовательно, могут изменяться внутри этих процедур.

Теперь настал момент подумать о том, каким способом в программе будет храниться и использоваться информация о текущем состоянии игры. Судя по всему, нам понадобятся хотя бы две переменные: в одной, назовем ее **NROW**, будет содержаться число рядов фишек, в другой (**NCOL**) — количество фишек в каждом ряду. Переменная **NROW** содержит одно целое положительное число, поэтому ее тип должен быть **INTEGER**. В переменной **NCOL** должно быть не менее **NROW** целых чисел, т.е. ее тип — это массив целых чисел. Поскольку в программе предусмотрена возможность изменения условий игры самим игроком, переменная **NROW** может меняться от партии к партии. В соответствии с этим должна была бы меняться и длина массива **NCOL**. Однако в Турбо Паскале нельзя использовать массивы, длина которых меняется динамически, т.е. в процессе работы программы. Эта длина должна определяться статически (на этапе компиляции) и не может меняться в работающей программе. Значит, понадобится массив достаточно большой длины, чтобы его хватило на все случаи. На экране одновременно можно отобразить максимум 25 строк по 80 символов в каждой строке. Однако использовать все строчки экрана как возможные ряды фишек вряд ли целесообразно: во-первых, сама игра при большом количестве рядов становится неинтересной, так как игрок не сможет проанализировать в уме все варианты ходов; во-вторых, на экране не останется места для вывода другой информации. Будем считать, что максимальное количество рядов фишек не должно превышать 14. Укажем это константой **MAXROW** — теперь, если Вы захотите назначить другое максимальное количество рядов, понадобится изменить значение этой константы и перекомпилировать программу. Именно таким способом программам придается дополнительная гибкость: Вы сосредоточиваете в

нескольких константах параметры, которые выбраны Вами произвольно и которые Вы или кто-то другой, возможно, захочет изменить. Все размеры массивов или другие особенности программной реализации следует определять через эти константы, тогда процедура переделки программы предельно упростится.

С учетом сказанного назовем следующие глобальные константы и переменные:

```
const
    MAXROW = 14;           {Максимальное количество рядов}
    MAXCOL = 20;           {Максимальное количество фишек в ряду}

type
    ColType = array [1..MAXROW] of Integer;

var
    exit   : Boolean;       {Признак окончания работы}
    change : Boolean;       {Признак изменения условий игры}
    nrow   : Integer;       {Количество рядов}
    ncol   : ColType;       {Максимальное колич-во фишек по рядам}
    col    : ColType;       {Текущее количество фишек по рядам}
```

Константа MAXCOL не участвует в формировании массивов, она будет использоваться для контроля горизонтальных размеров игрового поля. Поэтому она, а также пять переменных сделаны глобальными. Если считать, что начальная раскладка фишек соответствует схеме 3-4-5, то можно написать такой окончательный вариант главной программы:

```
Uses CRT; {Подключение библиотеки дополнительных
           процедур и функций для управления экраном}

const
    MAXROW = 14;           {Максимальное количество рядов}
    MAXCOL = 20;           {Максимальное количество фишек в ряду}

type
    ColType = array [1..MAXROW] of Integer;

var
    exit   : Boolean;       {Признак окончания работы}
    change : Boolean;       {Признак изменения условий игры}
    nrow   : Integer;       {Количество рядов}
    ncol   : ColType;       {Максимальное колич-во фишек по рядам}
    col    : ColType;       {Текущее количество фишек по рядам}
    -----}

Procedure Prepare;
{Готовит экран к игре}
begin {Prepare}
end; {Prepare}
    -----}

Procedure GetPlayerMove;
{Получает, контролирует и отображает ход игрока}
begin {GetPlayerMove}
end; {GetPlayerMove}
```

```

{-----}
Procedure SetOwnerMove;
{Находит и отображает очередной ход программы}
begin {SetOwnerMove}
end; {SetOwnerMove}
{-----}
begin {Главная программа}
  nrow      := 3;           {Готовим игру... }
  ncol[1]   := 3;           { на поле из трех }
  ncol[2]   := 4;           { рядов фишек }
  ncol[3]   := 5;           { по схеме 3-4-5.}
  repeat           {Цикл изменения условий игры}
  Prepare;         {Подготовить экран}
    repeat           {Игровой цикл}
    GetPlayerMove; {Получить ход пользователя}
    if not (exit or change) then
      SetOwnerMove {Определить собственный ход}
    until exit or change
  until exit
end.

```

Приступим к конструированию процедуры PREPARE. В ходе ее работы формируется значение переменной COL, соответствующее начальной раскладке фишек, и выводится информация о правилах игры. Чтобы было понятнее дальнейшее описание программной реализации, на рис. 2.4 показан вид экрана в начальном состоянии игры.

Процедура начинает свою работу с очистки экрана от имеющейся на нем информации. Это достигается обращением к стандартной процедуре без параметров CLRSCR. Затем выводятся три строчки с названием игры и кратким описанием ее правил. Кроме того, слева и справа на экране формируются заголовки для двух колонок цифр, в которых затем будут отображаться номер ряда (слева) и текущее количество фишек в ряду (справа). Эта информация поможет игроку сообщить программе свой ход. Для размещения информации на нужных участках экрана используется процедура GOTOXY(X,Y), с помощью которой курсор перемещается нужным образом. Параметры X и Y этой процедуры задают новые координаты курсора. Начало координат соответствует точке (1,1) и размещается в левом верхнем углу экрана, поэтому горизонтальная координата увеличивается слева направо, а вертикальная — сверху вниз.

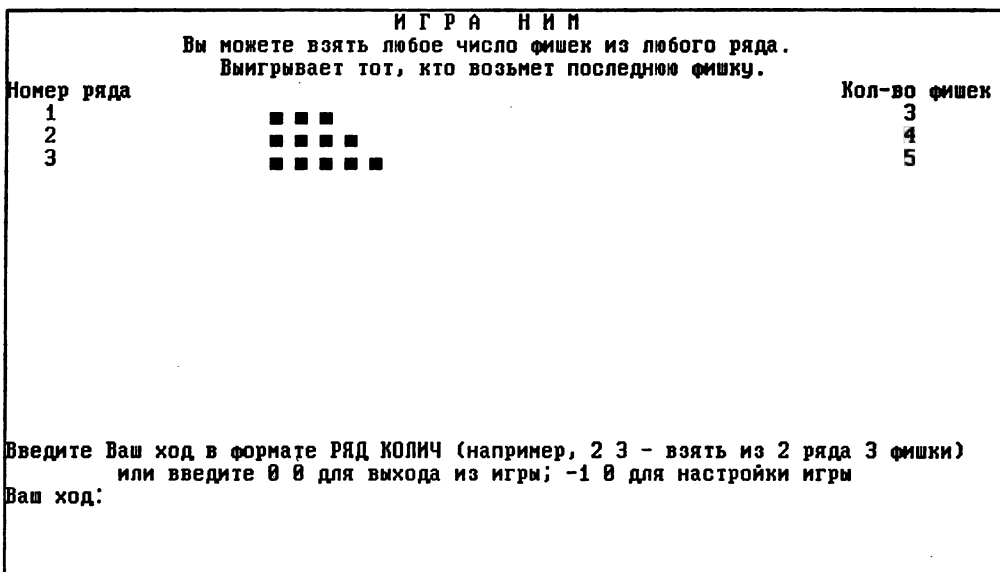


Рис.2.4. Вид экрана в начале игры ним

```

Procedure Prepare;
{Подготовка данных и экрана к игре}
const
Header0 = 'И Г Р А   Н И М';
Header1 = 'Вы можете взять любое число фишек из любого ряда.';
Header2 = 'Выигрывает тот, кто возьмет последнюю фишку.';
Header3 = 'Номер ряда';
Header4 = 'Количество фишек';
var
  i : Integer;
begin {Prepare}
  ClrScr;                                {Очищаем экран}
  {Выводим строки заголовка;}
  GotoXY((80-Length(Header0)) div 2,1);
  Write(Header0);
  GotoXY((80-Length(Header1)) div 2,2);
  Write(Header1);
  GotoXY((80-Length(Header2)) div 2,3);
  Writeln(Header2);
  Write(Header3);
  GotoXY(80-Length(Header4),4);
  Write(Header4);
  {Готовим начальную раскладку;}
  for i := 1 to nrow do
    col[i] := ncol[i]
end; {Prepare}

```

Для вывода верхних строк строго посередине экрана используется задание горизонтальной координаты курсора для процедуры GotoXY как половины от разницы между полной длиной экрана (80 позиций) и длиной выводимой строки (определяется с помощью функции LENGTH).

В процедуре GetPlayerMove осуществляются ввод, контроль и отображение на экране очередного хода игрока. Предварительно нужно показать игроку текущее состояние игрового поля. Поскольку поле будет обновляться как минимум дважды (после хода игрока и после хода программы), действия, связанные с изображением поля на экране, следует вынести в отдельную процедуру. Назовем ее ShowField и займемся ее реализацией.

Судя по всему, нам понадобится организовать цикл; в ходе цикла для каждого ряда игрового поля будет выведена строка, в левой части которой указывается номер ряда, в правой — текущее количество фишек в нем, а посередине выводятся символы, имитирующие фишки. В принципе, можно выбрать любой символ ПК для обозначения фишки, например, X или O. Я предпочел воспользоваться символом псевдографики с кодом 220: этот символ представляет собой небольшой квадратик и легко ассоциируется с фишкой.

Procedure ShowField;

{ Отображает на экране текущее состояние игрового поля }

const

FISH = #220;	<i>{ Символ-указатель фишки }</i>
X0 = 4;	<i>{ Левая колонка номеров рядов }</i>
X1 = 72;	<i>{ Правая колонка количества фишек }</i>
X = 20;	<i>{ Левый край игрового поля }</i>

var

i, j : Integer;

begin {ShowField}

for i := 1 to nrow **do**

begin

GotoXY(X0, i+4);

Write(i);

{ Номер ряда }

GotoXY(X1, i+4);

Write(col[i]:2);

{ Количество фишек в ряду }

for j := 1 to ncol[i] **do**

{ Вывод ряда фишек: }

begin

GotoXY(X+2*j, i+4);

if j[i] **then**

Write(FISH)

else

Write('.')

end

end

end; {ShowField}

Символы FISH (квадратики) выводятся через одну позицию, чтобы не сливались на экране. В те позиции, в которых ранее стояли уже снятые с поля фишки, выводится точка.

Теперь вернемся к процедуре GETPLAYERMOVE. При вводе любого очередного хода игрок должен задать два целых числа X1 и X2. Первое из них указывает номер ряда, а второе — количество фишек, которые игрок хочет забрать из этого ряда. Программа должна проконтролировать правильность задания этих чисел: X1 должно указывать непустой ряд, X2 не может превышать количество фишек в этом ряду. Кроме того, мы должны условиться о двух особых случаях:

- пользователь больше не хочет играть и дает команду завершить работу программы;
- пользователь хочет изменить условия игры.

Пусть ввод числа X1=0 означает команду выхода из программы, а X1=-1 — команду изменения условий игры. Тогда можно написать такой начальный вариант процедуры:

Procedure GetPlayerMove;

{Получает, контролирует и отображает ход игрока}

var

correctly : Boolean; {Признак правильности сделанного хода}

x1,x2 : Integer; {Вводимый ход}

begin {GetPlayerMove}

{Показываем начальное состояние игрового поля}

ShowField;

{Сообщаем игроку правила ввода хода}

repeat

{Приглашаем игрока ввести ход}

ReadLn(x1,x2); {Вводим очередной ход}

exit := x1=0; {Контроль команды выхода}

change := x1=-1; {Контроль команды изменения}

if not (exit **or** change) **then**

{Проверить правильность хода и установить нужное значение переменной CORRECTLY. Если ход правильный, сделать нужные изменения в раскладке фишек и показать поле.}

else

correctly := true {Случай EXIT или CHANGE}

until correctly;

if change **then**

{ Изменить условия игры }

end; {GetPlayerMove}

В этом варианте в процедуре GetPlayerMove нет описания процедуры SHOWFIELD. Сделано это не случайно: процедура ShowField может понадобиться также и при реализации процедуры SetOwnerMove, поэтому она должна быть глобальной по отношению и к GetPlayerMove, и к SetOwnerMove, т.е. ее описание должно в тексте программы предшествовать описаниям двух использующих ее процедур.

Действия

```

    { Сообщить игроку правила ввода хода },
    { Пригласить игрока ввести ход }

```

и

```

{Проверить правильность хода и установить нужное значение
переменной Correctly. Если ход правильный, сделать нужные
изменения в раскладке фишек и показать поле.}

```

не очень сложны в реализации, поэтому их можно осуществить непосредственно в теле процедуры GETPLAYERMOVE. Иное дело — изменение условий игры. Это действие полезно реализовать в отдельной процедуре GETCHANGE. С учетом этого второй вариант процедуры GETPLAYERMOVE примет такой вид:

Procedure GetPlayerMove;

{Получает, контролирует и отображает ход игрока}

const

```

TEXT1 = 'Введите Ваш ход в формате РЯД КОЛИЧ ';
TEXT01= '(например, 2 3 - взять из 2 ряда 3 фишки)';
TEXT2 = 'или введите 0 0 для выхода из игры; ';
TEXT02= '-1 0 для настройки игры';
TEXT3 = 'Ваш ход:          ';
Y = 20; {Номер строки для вывода сообщений}

```

var

```

correctly : Boolean; {Признак правильности сделанного хода}
x1,x2 : Integer;     {Вводимый ход}

```

{-----}

Procedure GetChange;

{Устанавливает новую настройку игры (количество рядов и количество фишек в каждом ряду)}

begin {GetChange}

end; {GetChange}

{-----}

begin {GetPlayerMove}

ShowField; {Показываем начальное состояние поля}

{Сообщить игроку правила ввода хода:}

GotoXY((80-Length(TEXT1+TEXT01)) **div** 2,Y);

Write(TEXT1+TEXT01);

GotoXY((80-Length(TEXT2+TEXT02)) **div** 2,Y+1);

Write(TEXT2+TEXT02);

repeat

{Пригласить игрока ввести ход:}

GotoXY(1,Y+2);

Write(TEXT3); {Выводим приглашение и
стираем предыдущий ход}

GotoXY(WhereX-16,Y+2); {Курсор влево на 16 позиций}

readln(x1,x2); {Вводим очередной ход}

exit := x1=0; {Контроль команды выхода}

change := x1=-1; {Контроль команды изменения}


```

if not (exit or change) then
  begin
    correctly := (x1 > 0) and (x1 <= nrow) and
                  (x2 <= col[x1]) and (x2 > 0);
    if correctly then
      begin
        col[x1] := col[x1]-x2; {Ход правильный;}
        ShowField {Изменяем раскладку фишек}
                  {Показываем поле}
      end
    else
      write(#7) {Ход неправильный: дать звуковой сигнал}
    end
  else
    correctly := true {Случай EXIT или CHANGE}
  until correctly;
  if change then
    GetChange
end; {GetPlayerMove}

```

Обратите внимание: константа

```
TEXT3 = 'Ваш ход:                ';
```

имеет длинный «хвост» из пробелов (их 17), поэтому после вывода этого приглашения курсор возвращается влево на 16 позиций оператором

```
GotoXY(WhereX-16,Y+2); {курсор влево на 16 позиций}
```

(функция WHEREX возвращает текущую горизонтальную координату курсора, а функция WHEREY — его вертикальную координату). Сделано это для того, чтобы в случае, если игрок ввел неверный ход и программа повторяет вывод приглашения, пробелы в константе TEXT3 затерли бы строку предыдущего ввода.

Чтобы завершить создание процедуры GETPLAYERMOVE, нужно спроектировать процедуру GETCHANGE, в которой осуществляется изменение условий игры. Я привожу текст этой процедуры без пояснений и приглашаю Вас самостоятельно разобраться в том, как она работает:

Procedure GetChange;

'Устанавливает новую настройку игры (количество рядов и количество фишек в каждом ряду)

const

```

t1 = 'Н А С Т Р О Й К А   И Г Р Ы';
t2 = '(ввод количества рядов и количества '+'
      'фишек в каждом ряду)';

```

var

```

correctly : Boolean;
i : Integer;

```

```

begin {GetChange}
  ClrScr;

```

```

GotoXY((80-Length(t1)) div 2,1);
Write(t1);
GotoXY((80-Length(t2)) div 2,2);
Write(t2);
repeat
  GotoXY(1,3);
  Write('Введите количество рядов (максимум ',
        MAXROW,'): ');
  GotoXY(WhereX-6,WhereY);
  ReadLn(nrow);
  correctly := (nrow <= MAXROW) and (nrow > 1);
  if not correctly then
    write(#7)
until correctly;
for i := 1 to nrow do
  repeat
    GotoXY(1,i+3);
    Write(' ряд ',i,', количество фишек (максимум ',
          MAXCOL,'): ');
    GotoXY(WhereX-6,WhereY);
    ReadLn(ncol[i]);
    correctly := (ncol[i] <= MAXCOL) and (ncol[i] > 0);
    if not correctly then
      write(#7)
  until correctly
end; {GetChange}

```

Переходим к конструированию процедуры SETOWNERMOVE, в которой программа должна проконтролировать текущую ситуацию на игровом поле и выбрать собственный ход. Работа процедуры начинается с подсчета числа непустых рядов. В зависимости от этого подсчета реализуются следующие действия:

- если все ряды пусты, значит предыдущим ходом игрок забрал последнюю фишку и он победил; нужно поздравить его с победой, усложнить игру и предложить сыграть еще раз;
- если есть только один непустой ряд, то очередной ход программы очевиден — забрать все фишки, что означает победу машины: сообщить об этом и предложить сыграть еще раз;
- если осталось два или более непустых ряда, выбрать собственный ход на основе оптимальной стратегии.

Начальный вариант процедуры:

Procedure SetOwnerMove;

'Находит и отображает очередной ход программы'
 {-----}

Function CheckField : Integer;

'Проверяет состояния игры. Возвращает 0, если нет ни одной фишки (победа игрока), 1 - есть один ряд (победа машины)'

```

и - количество непустых рядов в остальных случаях}
begin {CheckField}
end; {CheckField}
{-----}
Procedure PlayerVictory;
{Поздравить игрока с победой и усложнить игру}
begin {PlayerVictory}
end; {PlayerVictory}
{-----}
Procedure OwnVictory;
{Победа машины}
begin {OwnVictory}
end; {OwnVictory}
{-----}
Procedure ChooseMove;
{Выбор очередного хода}
begin {ChooseMove}
end; {ChooseMove}
{-----}
begin {SetOwnerMove}
  case CheckField of {Проверяем количество непустых рядов}
    0 : PlayerVictory; {Все ряды пусты - победа игрока}
    1 : OwnVictory;    {Один непустой ряд - победа машины}
  else
    ChooseMove;        {Выбираем очередной ход}
  end; {case}
end; {SetOwnerMove}

```

Функция CHECKFIELD и процедуры PLAYERVICTORY и OWNVICTORY достаточно просты и их текст помещается без каких-либо пояснений в окончательный вариант программы (см. прил.5.3). Отмечу лишь, что в случае победы игрока нет смысла повторять партию заново с той же самой раскладкой фишек. Поэтому игра усложняется: в исходную раскладку добавляется еще по одной фишке в каждый ряд.

В процедуре CHOOSEMOVE анализируется позиция и выбирается очередной ход программы. Описание оптимальной стратегии уже приводилось выше. Действия программы заключаются в поиске первого слева (старшего) двоичного разряда, для которого сумма чисел нечетная. Если такой разряд не обнаружен, то текущая позиция безопасна для игрока, а значит любой ход программы сделает ее опасной. В этом случае для программы не существует оптимального выбора и она лишь убирает одну фишку из любого непустого ряда. Такая тактика означает пассивное ожидание ошибки игрока.

Если обнаружен разряд i с нечетной суммой, программа приступает к реализации оптимальной стратегии и тогда игрок обречен на поражение. Для выбора ряда, из которого следует взять фишки, программа просматривает последовательно все ряды и отыскивает тот ряд j , количество фишек в котором (в двоичном представлении) дает единицу в разряде i .

Значение этого разряда для количества фишек в ряду j заменяется нулем. Затем программа продолжает подсчет суммы для оставшихся младших разрядов. Если в каком-либо из них вновь обнаружена нечетность, значение этого разряда для количества фишек в ряду j инвертируется, т.е. 0 заменяется на 1, а 1 на 0. Например, если двоичные представления числа фишек и четности сумм таковы:

число фишек в ряду j :	01001
четность сумм :	01011

(единицей указаны разряды с нечетными суммами), то в результате этой операции получим:

число фишек в ряду j :	00010
четность сумм :	00000

Таким образом, в исходном состоянии в ряду j было $1001 = 9$ фишек, безопасная позиция требует, чтобы в ряду осталось $0010 = 2$ фишки, следовательно, из него нужно забрать $9 - 2 = 7$ фишек.

Окончательный вариант программы представлен в прил.5.3. Попробуйте разобраться в ее деталях самостоятельно.

В программной реализации алгоритма широко используется то обстоятельство, что Ваш компьютер, как и все остальные вычислительные машины, работает с числами, представленными в двоичной системе счисления. Поэтому для получения двоичного представления числа в процедуре BITFORM оно проверяется на четность с помощью стандартной функции ODD, затем сдвигается вправо на один двоичный разряд (операция SHR), вновь осуществляется проверка на четность и т.д. до тех пор, пока не будут проверены все разряды. Максимальное число двоичных разрядов, достаточное для двоичного представления количества фишек в ряду MAXCOL=63, задается константой BIT=6.

Для получения суммы двоичных разрядов в процедуре CHOOSEMOVE используется суммирование разрядов по модулю 2 с помощью операции XOR. Такое суммирование дает 0, если количество единиц четное или равно нулю, и 1 — если нечетное. В этой же процедуре для инверсии двоичного разряда применяется оператор

```
if nbit[i] = 1 then
  ncbit[j,i] := ord(ncbit[j,i]=0); {Инверсия разрядов},
```

в котором используется соглашение о внутреннем представлении логических величин в Турбо Паскале: 0 соответствует FALSE, а 1 — TRUE.

Глава 3

ЭЛЕМЕНТЫ ЯЗЫКА

3.1. АЛФАВИТ

Алфавит языка Турбо Паскаль включает буквы, цифры, шестнадцатеричные цифры, специальные символы, пробелы и зарезервированные слова.

Буквы — это буквы латинского алфавита от *a* до *z* и от *A* до *Z*, а также знак подчеркивания *_* (код ASCII 95). В Турбо Паскале нет различия между прописными и строчными буквами алфавита, если только они не входят в символьные и строковые выражения.

Цифры — арабские цифры от 0 до 9.

Каждая *шестнадцатеричная цифра* имеет значение от 0 до 15. Первые 10 значений обозначаются арабскими цифрами 0...9, остальные шесть — латинскими буквами *A...F* или *a...f*.

Специальные символы Турбо Паскаля — это символы

+ - * / = , ' . : ; < > [] () { } ^ @ \$ #

К специальным символам относятся также следующие *пары символов*:

<> <= >= := (* *) (. .)

В программе эти пары символов нельзя разделять пробелами, если они используются как знаки операций отношения или ограничители комментария. Символы *(. и .)* могут употребляться соответственно вместо *[и]*.

Особое место в алфавите языка занимают *пробелы*, к которым относятся любые символы ASCII в диапазоне кодов от 0 до 32. Эти символы рассматриваются как ограничители идентификаторов, констант, чисел, зарезервированных слов. Несколько следующих друг за другом пробелов считаются одним пробелом (последнее не относится к строковым константам).

В Турбо Паскале имеются следующие *зарезервированные слова*:

and	end	nil	shr
asm	file	not	string
array	for	object	then
begin	function	of	to
case	goto	or	type
const	if	packed	unit
constructor	implementation	procedure	until
destructor	in	program	uses
div	inline	record	var
do	interface	repeat	while
downto	label	set	with
else	mod	shl	xor

Зарезервированные слова не могут использоваться в качестве идентификаторов.

Стандартные директивы первоначально связаны с некоторыми стандартными объявлениями в программе. К ним относятся:

absolute	far	near
assembler	forward	private
external	interrupt	virtual

Как и зарезервированные слова, стандартные директивы в окне редактора Турбо Паскаля выделяются цветом, тем не менее Вы можете переопределить любую стандартную директиву, т.е. объявить одноименный идентификатор. Стандартные директивы **PRIVATE** и **VIRTUAL** действуют только в пределах объявления объектов.

3.2. ИДЕНТИФИКАТОРЫ

Идентификаторы в Турбо Паскале — это имена констант, переменных, меток, типов, объектов, процедур, функций, модулей, программ и полей в записях. Идентификаторы могут иметь произвольную длину, но значащими (уникальными в области определения) являются только первые 63 символа.

Идентификатор всегда начинается буквой, за которой могут следовать буквы и цифры. Напомню, что буквой считается также символ подчеркивания, поэтому идентификатор может начинаться этим символом и даже состоять только из одного или нескольких символов подчеркивания. Пробелы и специальные символы алфавита не могут входить в идентификатор.

Примеры правильных идентификаторов:

```
a
ALPHA
MyProgramIsBestProgram
date_27_sep_39
external
_beta
```

Примеры неправильных идентификаторов:

```
1Program    {начинается цифрой}
block#1     {содержит специальный символ}
My Prog     {содержит пробел}
mod         {зарезервированное слово}
```

3.3. КОНСТАНТЫ

В качестве констант в Турбо Паскале могут использоваться целые, вещественные и шестнадцатеричные числа, логические константы, символы, строки символов, конструкторы множеств и признак неопределенного указателя **NIL**.

Целые числа записываются со знаком или без него по обычным правилам и могут иметь значение от -2147483648 до $+2147483647$. Следует

учесть, что, если целочисленная константа выходит за указанные границы, компилятор дает сообщение об ошибке. Такие константы должны записываться с десятичной точкой, т.е. определяться как вещественные числа.

Вещественные числа записываются со знаком или без него с использованием десятичной точки и/или экспоненциальной части. Экспоненциальная часть начинается символом *e* или *E*, за которым могут следовать знаки «+» или «-» и десятичный порядок. Символ *e* (*E*) означает десятичный порядок и имеет смысл «умножить на 10 в степени». Например,

3.14E5 - 3.14 умножить на 10 в степени 5;

-17e-2 - минус 17 умножить на 10 в степени минус 2.

Если в записи вещественного числа присутствует десятичная точка, перед точкой и за ней должно быть хотя бы по одной цифре. Если используется символ экспоненциальной части *e* (*E*), за ним должна следовать хотя бы одна цифра десятичного порядка.

Шестнадцатеричное число состоит из шестнадцатеричных цифр, которым предшествует знак доллара \$ (код 36 в ASCII). Диапазон шестнадцатеричных чисел — от \$00000000 до \$FFFFFFFF.

Логическая константа — это либо слово FALSE (ложь), либо слово TRUE (истина).

Символьная константа — это любой символ ПК, заключенный в апострофы:

'z' - символ z;

'ф' - символ ф.

Если необходимо записать собственно символ апострофа, он удваивается:

'' - символ ' (апостроф).

Допускается использование записи символа путем указания его внутреннего кода, которому предшествует символ # (код 35), например:

#97 - символ а;

#90 - символ Z;

#39 - символ ';

#13 - символ CR.

Строковая константа — любая последовательность символов (кроме символа CR — возврат каретки), заключенная в апострофы. Если в строке нужно указать сам символ апострофа, он удваивается, например:

'Это - строка символов';

'That's string'.

Строка символов может быть пустой, т.е. не иметь никаких символов в обрамляющих ее апострофах. Строку можно составлять из кодов нужных символов с предшествующими каждому коду символами #, например, строка #83#121#109#98#11#108 эквивалентна строке 'Symbol'.

Наконец, в строке можно чередовать части, записанные в обрамляющих апострофах, с частями, записанными кодами. Таким способом можно

вставлять в строки любые управляющие символы, в том числе и символ CR (код 13), например:

```
#7'Ошибка !'#13'Нажмите любую клавишу ...'#7 .
```

Конструктор множества — список элементов множества, обрамленный квадратными скобками, например:

```
[1,2,4..7,12]
[blue, red]
[]
[true]
```

В отличие от стандартного Паскаля, в Турбо Паскале разрешается в объявлении констант использовать произвольные *выражения*, операндами которых могут быть ранее объявленные нетипизированные константы, имена типов и объектов, а также следующие функции от них:

abs	lo	ptr	swap
chr	odd	round	trunc
hi	ord	sizeof	
length	pred	succ	

Например:

const

```
MaxReal = MaxInt div SizeOf(real);
NumChars = ord('Z') - ord('a') + 1;
Ln10 = 2.302585092994;
Ln10R = 1 / Ln10;.
```

3.4. ВЫРАЖЕНИЯ

Основными элементами, из которых конструируется исполняемая часть программы, являются константы, переменные и обращения к функциям. Каждый из этих элементов характеризуется своим значением и принадлежит к какому-либо типу данных. С помощью знаков операций и скобок из них можно составлять *выражения*, которые фактически представляют собой правила получения новых значений.

Частным случаем выражения может быть просто одиночный элемент, т.е. константа, переменная или обращение к функции. Значение такого выражения имеет, естественно, тот же тип, что и сам элемент. В более общем случае выражение состоит из нескольких элементов (операндов) и знаков операций, а тип его значения определяется типом операндов и видом примененных к ним операций.

Примеры выражений:

```
y
21
(a + b) * c
sin(t)
a > 2
not Flag and (a = b)
```


NIL

[1, 3..7] * set1

3.5. ОПЕРАЦИИ

В Турбо Паскале определены следующие операции:

унарные	not, @;
мультипликативные	*, /, div, mod, and, shl, shr;
аддитивные	+, -, or, xor;
отношения	=, <>, <, >, <=, >=, in.

Приоритет операций убывает в указанном порядке, т.е. наивысшим приоритетом обладают унарные операции, низшим — операции отношения. Порядок выполнения нескольких операций равного приоритета устанавливается компилятором из условия оптимизации кода программы и не обязательно слева направо. При исчислении логических выражений операции равного приоритета всегда вычисляются слева направо, причем будут вычисляться все или только достаточные операции в зависимости от установленной в среде Турбо Паскаля опции *OPTIONS/COMPILER/COMPLETE BOOLEAN EVAL*: при установленном значении этой опции вычисляются все операции отношения, при неустановленном — только те, которые достаточны для получения результата.

Это обстоятельство необходимо учитывать при использовании операций отношения с функциями, в которых изменяются глобальные переменные или параметры, передаваемые по имени, например:

```
Function AddI(var x: Integer): Integer;
begin {AddI}
  inc(x);
  AddI := x
end {AddI};
var
  a, b : Integer;
begin {main}
  if (a > b) or (AddI(a) > 100) then b := a;
  .....
```

При выполнении этого фрагмента значение переменной A будет зависеть от настройки опции: если опция активизирована, значение A всегда наращивается на 1, если не активизирована — только в случае $A \leq B$.

Правила использования операций с операндами различного типа приводятся в табл. 3.1.

Таблица 3.1

Опера- ция	Действие	Тип операндов	Тип результата
not	Отрицание	Логический	Логический
not	То же	Любой целый	Тип операнда
@	Адрес	Любой	Указатель

*	Умножение	Любой целый	Наименьший целый
*	То же	Любой вещественный	Extended
*	Пересечение множеств	Множественный	Множественный
/	Деление	Любой вещественный	Extended
div	Целочисленное деление	Любой целый	Наименьший целый
mod	Остаток от деления	То же	То же
and	Логическое И	Логический	Логический
and	То же	Любой целый	Наименьший целый
shl	Левый сдвиг	То же	То же
shr	Правый сдвиг	То же	То же
+	Сложение	То же	То же
+	То же	Любой вещественный	Extended
+	Объединение множеств	Множественный	Множественный
+	Сцепление строк	Строковый	Строковый
-	Вычитание	Любой целый	Наименьший целый
-	То же	Любой вещественный	Extended
or	Логическое ИЛИ	Логический	Логический
or	То же	Любой целый	Наименьший целый
=	Равно	Любой простой или строковый	Логический
<>	Не равно	То же	То же
<	Меньше	Логический	Логический
<=	Меньше или равно	То же	То же
>	Больше	То же	То же
>=	Больше или равно	То же	То же

При действиях с вещественным типом одним из операндов может быть значение любого целого типа. Результат операций имеет указанный в таблице тип EXTENDED только для установленного в среде Турбо Паскаля режима генерации кода, рассчитанного на арифметический сопроцессор или на его эмуляцию (см. прил.1). Если этот режим не установлен, результат будет иметь значение типа REAL.

Унарная операция @ применяется к операнду любого типа и возвращает результат типа POINTER (см. гл.6), в котором содержится адрес операнда. Пусть, например, задано описание

```

type
  TwoChar = array [1..2] of char;
var
  Int      : integer;
  TwoCharPtr : ^TwoChar;

```

Тогда оператор

```
TwoCharPtr := @Int;
```

приведет к тому, что в TwoCharPtr будет храниться адрес целочисленной переменной INT, которая может теперь интерпретироваться как совокупность двух символов. Поэтому возможен, например, такой оператор:

```
if TwoCharPtr^[1] = 'C' then ...
```

Если операция @ применяется к процедуре, функции или методу в объекте, ее результатом будет адрес точки входа в эту процедуру

(функцию, метод). Этот адрес можно использовать только в подпрограмме, написанной на ассемблере, или в фрагментах **INLINE**.

В Турбо Паскале определены следующие логические операции:

not - логическое НЕ;
and - логическое И;
or - логическое ИЛИ;
xor - исключительное ИЛИ.

Логические операции применимы к операндам целого и логического типов. Если операнды — целые числа, то результат логической операции есть тоже целое число, биты которого (двоичные разряды) формируются из битов операндов по правилам, указанным в табл. 3.2.

Таблица 3.2

**Логические операции над данными типа INTEGER
(поразрядно)**

Операнд 1	Операнд 2	not	and	or	xor
1	-	0	-	-	-
0	-	1	-	-	-
0	0	-	0	0	0
0	1	-	0	1	1
1	0	-	0	1	1
1	1	-	1	1	0

К логическим же в Турбо Паскале обычно относятся и две сдвиговые операции над целыми числами:

i shl j — сдвиг содержимого *i* на *j* разрядов влево; освободившиеся младшие разряды заполняются нулями;

i shr j — сдвиг содержимого *i* на *j* разрядов вправо; освободившиеся старшие разряды заполняются нулями.

В этих операциях *i* и *j* — выражения любого целого типа.

С помощью программы примера 3.1 можно вывести на экран результат применения логических операций к двум целым числам.

Пример 3.1

{Программа вводит два целых числа и печатает результат применения к ним логических операций. Для выхода из программы ввести Ctrl-Z и нажать Enter}

var

n,m : integer;

begin

while not EOF do

begin

Write('n,m = ');

ReadLn(n,m);

WriteLn(' not= ', not n, ' ', not m);

```

WriteLn(' and= ',n and m);
WriteLn(' or = ',n or m);
WriteLn(' xor= ',n xor m);
WriteLn(' shl= ',n shl m);
WriteLn(' shr= ',n shr m)

```

end

end.

В программе организуется ввод двух произвольных целых чисел и печать результата применения к ним всех логических операций. Для выхода из программы следует нажать *Ctrl-Z* и *Enter*.

Логические операции над логическими данными дают результат логического типа по правилам, указанным в табл. 3.3.

Таблица 3.3

Логические операции над данными типа Boolean					
Операнд 1	Операнд 2	not	and	or	xor
True	-	False	-	-	-
False	-	True	-	-	-
False	False	-	False	False	False
False	True	-	False	True	True
True	False	-	False	True	True
True	True	-	True	True	False

Операция отношения **IN** применяется к двум операндам. Первым (левым) операндом должно быть выражение любого порядкового типа, вторым — множество, состоящее из элементов того же типа, или идентификатор множественного типа. Операция дает TRUE, если левый операнд принадлежит множеству, например:

```

var
  c: char;
type
  digit = set of '0'..'9';
begin
  if c in digit then .....

```

3.6. СТРУКТУРА ПРОГРАММЫ

Структура любой программной единицы (программы, процедуры или функции) должна быть такой:

```

<Объявление программной единицы>
{Раздел описаний}
BEGIN
{Раздел исполняемых операторов}
END <символ конца программной единицы>

```

Здесь *<Объявление программной единицы>* — заголовок программы, процедуры или функции; заголовок программы можно опускать без каких-либо последствий для программы; для процедур и функций наличие заголовка обязательно;
<символ конца программной единицы> — символ «.» для программы или символ «;» — для процедуры и функции.

Любой из двух разделов программной единицы — раздел описаний или исполняемых операторов, или оба одновременно могут быть пустыми, т.е. не содержать никаких описаний или исполняемых операторов.

В разделе описаний должны содержаться описания всех идентификаторов, используемых в разделе исполняемых операторов. Исключением являются идентификаторы, определенные в интерфейсных частях программных модулей (библиотек), а также глобальные для процедуры или функции идентификаторы (см. гл. 8). Если программная единица использует идентификатор из интерфейсной части какого-либо модуля (см. гл. 9), в начале программы в предложении **USES** необходимо указать имя этого модуля. Последнее не относится к идентификаторам, определенным в стандартном модуле **SYSTEM**, т.е. имя этого модуля в предложении **USES** указывать не нужно. Более того, модуль **SYSTEM** считается предварительно объявленным, поэтому объявление

Uses System;

компилятор расценит как попытку двойного объявления модуля **SYSTEM** и даст соответствующее сообщение об ошибке.

В разделе описаний объявляются идентификаторы типов, объектов, констант, переменных, а также метки, процедуры и функции. Описанию типов и объектов должно предшествовать зарезервированное слово **TYPE**, описанию констант — **CONST**, переменных — **VAR** и меток — **LABEL**, на пример:

```
type
  DigType = set of '0'..'9';
  StrType = string [40];
const
  N = 100;
  EPS = 1e-9;
var
  x, y : real;
  st   : StrType;
label
  lb1, lb2;
```

В отличие от стандартного Паскаля разделы **TYPE**, **CONST**, **VAR**, **LABEL** могут следовать друг за другом в любом порядке и встречаться в разделе описаний сколько угодно раз.

Описание процедуры или функции заключается в указании заголовка этой процедуры (функции) и ее тела (подробнее см. в гл. 8).

Структура программных библиотек (модулей) описана в гл.9.

Глава 4

ТИПЫ ДАННЫХ

Любые данные, т.е. константы, переменные, значения функций или выражения, в Турбо Паскале характеризуются своими типами. *Тип* определяет множество допустимых значений, которые может иметь тот или иной объект, а также множество допустимых операций, которые применимы к нему. Кроме того, тип определяет также и формат внутреннего представления данных в памяти ПК.

Турбо Паскаль характеризуется разветвленной структурой типов данных (рис.4.1).

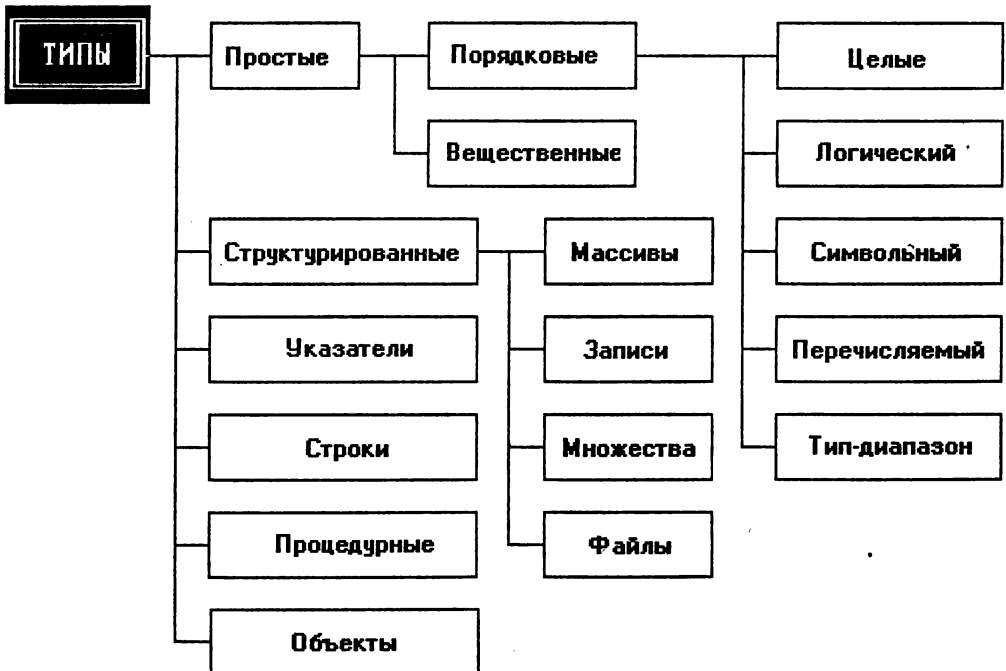


Рис.4.1. Структура типов данных

В Турбо Паскале предусмотрен механизм создания новых типов данных, благодаря чему общее количество типов, используемых в программе, может быть сколь угодно большим.

В этой главе приводится подробное описание всех типов, за исключением файлов и указателей, которые рассматриваются в следующих двух главах, а также процедурных типов, которые рассматриваются в гл.8, и объектов (гл.10).

4.1. ПРОСТЫЕ ТИПЫ

К простым типам относятся порядковые и вещественные типы.

Порядковые типы отличаются тем, что каждый из них имеет конечное число возможных значений. Эти значения можно определенным образом упорядочить (отсюда — название типов) и, следовательно, с каждым из них можно сопоставить некоторое целое число — порядковый номер значения.

Вещественные типы, строго говоря, тоже имеют конечное число значений, которое определяется форматом внутреннего представления вещественного числа. Однако количество возможных значений вещественных типов настолько велико, что сопоставить с каждым из них целое число (его номер) не представляется возможным.

4.1.1. Порядковые типы

К порядковым типам относятся (см. рис.4.1) целые, логический, символьный, перечисляемый и тип-диапазон. К любому из них применима функция $ORD(X)$, которая возвращает порядковый номер значения выражения X . Для целых типов функция $ORD(X)$ возвращает само значение X , т.е. $ORD(X) = X$ для X , принадлежащего любому целому типу. Применение $ORD(X)$ к логическому, символьному и перечисляемому типам дает положительное целое число в диапазоне от 0 до 1 (логический тип), от 0 до 255 (символьный), от 0 до 65535 (перечисляемый). Тип-диапазон сохраняет все свойства базового порядкового типа, поэтому результат применения к нему функции $ORD(X)$ зависит от свойств этого типа.

К порядковым типам можно также применять функции:

$PRED(X)$ — возвращает предыдущее значение порядкового типа (значение, которое соответствует порядковому номеру $ORD(X) - 1$), т.е.

$$ORD(PRED(X)) = ORD(X) - 1;$$

$SUCC(X)$ — возвращает следующее значение порядкового типа, которое соответствует порядковому номеру $ORD(X) + 1$, т.е.

$$ORD(SUCC(X)) = ORD(X) + 1.$$

Например, если в программе определена переменная

```
var
  c : char;
begin
  c := '5';
end.
```

то функция $PRED(C)$ вернет значение '4', а функция $SUCC(C)$ — значение '6'.

Если представить себе любой порядковый тип как упорядоченное множество значений, возрастающих слева направо и занимающих на числовой оси некоторый отрезок, то функция $PRED(X)$ не определена для левого, а $SUCC(X)$ — для правого конца этого отрезка.

Целые типы. Диапазон возможных значений целых типов зависит от их внутреннего представления, которое может занимать один, два или четыре

байта. В табл. 4.1 приводится название целых типов, длина их внутреннего представления в байтах и диапазон возможных значений.

Таблица 4.1

Целые типы		
Название	Длина, байт	Диапазон значений
Byte	1	0...255
ShortInt	1	-128...+127
Word	2	0...65535
Integer	2	-32768...+32767
LongInt	4	-2 147 483 648...+2 147 483 647

При использовании процедур и функций с целочисленными параметрами следует руководствоваться «вложенностью» типов, т.е. везде, где может использоваться *WORD*, допускается использование *BYTE* (но не наоборот), в *LONGINT* «входит» *INTEGER*, который, в свою очередь, включает в себя *SHORTINT*.

Перечень процедур и функций, применимых к целочисленным типам, приведен в табл.4.2. Буквами *b, s, w, i, l* обозначены выражения соответственно типа *BYTE, SHORTINT, WORD, INTEGER* и *LONGINT*, *x* – выражение любого из этих типов; буквы *vb, vs, vw, vi, vl, vx* обозначают переменные соответствующих типов. В квадратных скобках указывается не-обязательный параметр.

Таблица 4.2

Стандартные процедуры и функции, применимые к целым типам		
Обращение	Тип результата	Действие
abs(x)	x	Возвращает модуль x
chr(b)	Char	Возвращает символ по его коду
dec(vx[,i])	-	Уменьшает значение vx на i, а при отсутствии i – на 1
inc(vx[,i])	-	Увеличивает значение vx на i, а при отсутствии i – на 1
Hi(i)	Byte	Возвращает старший байт аргумента
Hi(w)	То же	То же
Lo(i)	"	Возвращает младший байт аргумента
Lo(w)	"	То же
odd(l)	Boolean	Возвращает True, если аргумент – нечетное число
Random(w)	Как у параметра	Возвращает псевдослучайное число, равномерно распределенное в диа-

		пазоне 0... (w-1)
sqr(x)	x	Возвращает квадрат аргумента
swap(i)	Integer	Меняет местами байты в слове
swap(w)	Word	

При действиях с целыми числами тип результата будет соответствовать типу операндов, а если операнды относятся к различным целым типам, — типу того операнда, который имеет максимальную мощность (максимальный диапазон значений). Возможное переполнение результата никак не контролируется, что может привести к недоразумениям, например:

```
var
  a : integer;
  x, y : real;
begin
  a := 32767;           {Максимально возможное значение типа
                        INTEGER}
  x := a + 2;           {Переполнение при вычислении
                        этого выражения!}
  y := LongInt(a)+2;    {Переполнения нет после приведения
                        переменной к более мощному типу}
  WriteLn(x:10:0, y:10:0)
end.
```

В результате прогона программы получим

```
-32767      32769
```

Логический тип. Значениями логического типа может быть одна из предварительно объявленных констант *FALSE* (ложь) или *TRUE* (истина). Для них справедливы правила:

```
ord(false) = 0;
ord(true)  = 1;
false < true;
succ(false) = true;
pred(true)  = false.
```

Поскольку логический тип относится к порядковым типам, его можно использовать в операторе счетного типа, например:

```
var
  l : Boolean;
begin
  for l := False to True do ....
```

Символьный тип. Значением символьного типа является множество всех символов ПК. Каждому символу приписывается целое число в диапазоне 0...255. Это число служит кодом внутреннего представления символа, его возвращает функция *ORD*.

Для кодировки используется код *ASCII* (*American Standard Code for Information Interchange* — американский стандартный код для обмена

информацией). Это 7-битный код, т.е. с его помощью можно закодировать лишь 128 символов в диапазоне от 0 до 127. В то же время в 8-битном байте, отведенном для хранения символа в Турбо Паскале, можно закодировать в два раза больше символов в диапазоне от 0 до 255. Первая половина символов ПК с кодами 0...127 соответствует стандарту ASCII (табл. 4.3). Вторая половина символов с кодами 128...255 не ограничена жесткими рамками стандарта и может меняться на ПК разных типов (в прил.2 приведены некоторые распространенные варианты кодировки этих символов).

Таблица 4.3

Кодировка символов в соответствии со стандартом ASCII							
Код	Символ	Код	Символ	Код	Символ	Код	Символ
0	NUL	32	BL	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DEL	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~

31	US	63	?	95		127	□
----	----	----	---	----	--	-----	---

Символы с кодами 0...31 относятся к служебным кодам. Если эти коды используются в символьном тексте программы, они считаются пробелами. При использовании их в операциях ввода – вывода они могут иметь следующее самостоятельное значение:

Символ	Код	Значение
BEL	7	Звонок; вывод на экран этого символа сопровождается звуковым сигналом
HT	9	Горизонтальная табуляция; при выводе на экран смещает курсор в позицию, кратную 8, плюс 1 (9, 17, 25 и т.д.)
LF	10	Перевод строки; при выводе его на экран все последующие символы будут выводиться, начиная с той же позиции, но на следующей строке
VT	11	Вертикальная табуляция; при выводе на экран заменяется специальным знаком
FF	12	Прогон страницы; при выводе на принтер формирует страницу, при выводе на экран заменяется специальным знаком
CR	13	Возврат каретки; вводится нажатием на клавишу <i>Enter</i> (при вводе с помощью <i>READ</i> или <i>READLN</i> означает команду «Ввод» и в буфер ввода не помещается; при выводе означает команду «Продолжить вывод с начала текущей строки»)
SUB	26	Конец файла; вводится с клавиатуры нажатием <i>Ctrl-Z</i> ; при выводе заменяется специальным знаком
ESC	27	Конец работы; вводится с клавиатуры нажатием на клавишу <i>ESC</i> ; при выводе заменяется специальным знаком

К типу *CHAR* применимы операции отношения, а также встроенные функции:

CHR(B) – функция типа *CHAR*; преобразует выражение *B* типа *BYTE* в символ и возвращает его своим значением;

UPCASE(CH) – функция типа *CHAR*; возвращает прописную букву, если *CH* – строчная латинская буква, в противном случае возвращает сам символ *CH*, например:

```
var
  c1,c2: char;
begin
  c1 := UpCase('s');
  c2 := UpCase('ф');
  WriteLn(c1, ' ', c2)
end.
```

Так как функция *UPCASE* не обрабатывает кириллицу, в результате прогона этой программы на экран будет выдано

S ф

Перечисляемый тип. Перечисляемый тип задается перечислением тех значений, которые он может получать. Каждое значение именуется не –

которым идентификатором и располагается в списке, обрамленном круглыми скобками, например:

type

```
colors = (red, white, blue);
```

Применение перечисляемых типов делает программы нагляднее. Если, например, в программе используются данные, связанные с месяцами года, то такой фрагмент программы:

type

```
ТипМесяц = (январь, февраль, март, апрель, май, июнь, июль, август, сентябрь, октябрь, ноябрь, декабрь);
```

var

```
месяц : ТипМесяц;
```

begin

```
.....
```

```
if месяц = август then WriteLn('Хорошо бы поехать к морю!');
```

```
.....
```

end.

был бы, согласитесь, очень наглядным. Увы! В Турбо Паскале нельзя использовать кириллицу в идентификаторах, поэтому мы вынуждены писать так:

type

```
TypeMonth = (jan, feb, mar, may, jun, jul, aug, sep, oct, nov, dec);
```

var

```
month: TypeMonth;
```

begin

```
.....
```

```
if month = aug then WriteLn('Хорошо бы поехать к морю!');
```

```
.....
```

end.

Соответствие между значениями перечисляемого типа и порядковыми номерами этих значений устанавливается порядком перечисления: первое значение в списке получает порядковый номер 0, второе — 1 и т.д. Максимальная мощность перечисляемого типа составляет 65536 значений, поэтому фактически перечисляемый тип задает некоторое подмножество целого типа *WORD* и может рассматриваться как компактное объявление сразу группы целочисленных констант со значениями 0, 1 и т.д.

Использование перечисляемых типов повышает надежность программ благодаря возможности контроля тех значений, которые получают соответствующие переменные. Пусть, например, заданы такие перечисляемые типы:

type

```
colors = (black, red, white);
```

```
ordinal = (one, two, three);
```

```
days = (monday, tuesday, wednesday);
```

С точки зрения мощности и внутреннего представления все три типа эквивалентны:

```
ord(black)=0, ... , ord(white)=2,
ord(one)=0, ... , ord(three)=2,
ord(monday)=0, ... , ord(wednesday)=2.
```

Однако, если определены переменные

```
var
  col : colors;
  num : ordinal;
  day : days;
```

то допустимы операторы

```
col := black;
num := succ(two);
day := pred(tuesday);
```

но недопустимы

```
col := one;
day := black;
```

Как уже упоминалось, между значениями перечисляемого типа и множеством целых чисел существует однозначное соответствие, задаваемое функцией $ORD(X)$. В Турбо Паскале допускается и обратное преобразование: любое выражение типа *WORD* можно преобразовать в значение перечисляемого типа, если только значение целочисленного выражения не превышает мощности перечисляемого типа. Такое преобразование достигается применением автоматически объявляемой функции с именем перечисляемого типа (см. п. 4.4). Например, для рассмотренного выше объявления типов эквивалентны следующие присваивания:

```
col := one;
col := colors(0);
```

Разумеется, присваивание

```
col := 0;
```

будет недопустимым.

Переменные любого перечисляемого типа можно объявлять без предварительного описания этого типа, например:

```
var
  col: (black, white, green);
```

Тип-диапазон. Тип-диапазон есть подмножество своего базового типа, в качестве которого может выступать любой порядковый тип, кроме типа-диапазона.

Тип-диапазон задается границами своих значений внутри базового типа:

```
<мин.знач.>...<макс.знач.>
```

Здесь <мин.знач.> — минимальное значение типа-диапазона;

<макс.знач.> — максимальное его значение.

Например:

```
type
  digit   = '0'..'9';
  dig2    = 48 .. 57;
```

Тип-диапазон необязательно описывать в разделе *TYPE*, а можно указывать непосредственно при объявлении переменной, например:

```
var
  date : 1..31;
  month: 1..12;
  lchr : 'A'..'Z';.
```

При определении типа-диапазона нужно руководствоваться следующими правилами:

- два символа «..» рассматриваются как один символ, поэтому между ними недопустимы пробелы;
- левая граница диапазона не должна превышать его правую границу.

Тип-диапазон наследует все свойства своего базового типа, но с ограничениями, связанными с его меньшей мощностью. В частности, если определена переменная

```
type
  days    = (mo,tu,we,th,fr,sa,su);
  WeekEnd = sa .. su;
var
  w : WeekEnd;
begin
  .....
  w := sa;
  .....
end;
```

то *ORD(W)* вернет значение 5, в то время как *PRED(W)* приведет к ошибке.

В стандартную библиотеку Турбо Паскаля включены две функции, поддерживающие работу с типами-диапазонами:

HIGH(X) — возвращает максимальное значение типа-диапазона, к которому принадлежит переменная X;

LOW(X) — возвращает минимальное значение типа-диапазона.

Следующая короткая программа выведет на экран строку

-32768...32767

```
var k: Integer;
begin
  WriteLn(Low(k), '...', High(k));
end.
```

4.1.2. Вещественные типы

В отличие от порядковых типов, значения которых всегда сопоставляются с рядом целых чисел и, следовательно, представляются в ПК абсолютно точно, значения вещественных типов определяют произвольное число лишь с некоторой конечной точностью, зависящей от внутреннего формата вещественного числа.

Таблица 4.4

Длина, байт	Название	Количество значащих цифр	Диапазон десятичного порядка
6	real	11...12	-39...+38
8	double	15...16	-324...+308
10	extended	19...20	-4951...+4932
8	comp	19...20	$-2 \cdot 10^{63} + 1 \dots + 2 \cdot 10^{63} - 1$

Как видно из табл.7, вещественное число в Турбо Паскале занимает от 4 до 10 смежных байт и имеет следующую структуру в памяти ПК:

s	e	m
---	---	---

Здесь *s* — знаковый разряд числа; *e* — экспоненциальная часть; *m* — мантисса числа.

Мантисса *m* имеет длину от 23 (для *SINGLE*) до 63 (для *EXTENDED*) двоичных разрядов, что и обеспечивает точность 7...8 для *SINGLE* и 19...20 для *EXTENDED* десятичных цифр. Десятичная точка (запятая) подразумевается перед левым (старшим) разрядом мантиссы, но при действиях с числом ее положение сдвигается влево или вправо в соответствии с двоичным порядком числа, хранящимся в экспоненциальной части, поэтому действия над вещественными числами называют арифметикой с плавающей точкой (запятой).

Как видим, Турбо Паскаль характеризуется богатой гаммой вещественных типов, однако доступ к типам *SINGLE*, *DOUBLE* и *EXTENDED* возможен только при особых режимах компиляции. Дело в том, что эти типы рассчитаны на аппаратную поддержку арифметики с плавающей точкой и для их эффективного использования в состав ПК должен входить арифметический сопроцессор. Компилятор Турбо Паскаля позволяет создавать программы, работающие на любых ПК (с сопроцессором или без него) и использующие любые вещественные типы. Необходимая для этого настройка компилятора описана в прил.1. В процессе запуска Турбо Паскаль проверяет состав аппаратных средств и выявляет наличие или отсутствие сопроцессора.

В некоторых случаях бывает необходимо отключить автоконтроль. Для этого перед запуском Турбо Паскаля следует дать такую команду ДОС:

```
set 87=N
```

Команда

```
set 87=Y
```

напротив, включает автоконтроль — эта команда активна по умолчанию.

Отметим, что арифметический сопроцессор всегда обрабатывает числа в формате *EXTENDED*, а три других вещественных типа в этом случае по-лучаются простым усечением результатов до нужных размеров и при-меняются в основном для экономии памяти.

Например, если «машинное epsilon» (см. пример 2.6 в гл.2) вычисляется с помощью такой программы:

```
{ $N+, E+ }
type
  RealType = real;
var
  epsilon : RealType;
begin
  epsilon := 1;
  while 1+epsilon/2 > 1 do
    epsilon := epsilon/2;
  WriteLn(epsilon)
end.
```

то независимо от объявления типа *REALTYPE* (он может быть *SINGLE*, *REAL*, *DOUBLE* или *EXTENDED*) на печать будет выдан результат

1.08420217248550E-0019

что соответствует типу *EXTENDED*. Происходит это по той причине, что все операнды вещественного выражения $1 + \text{epsilon}/2$ в операторе *WHILE*, перед вычислением автоматически преобразуются к типу *EXTENDED*. Чтобы получить правильный результат (например, для типа *REALTYPE = REAL* он будет 9.09494701772928E-0013), программу не-обходимо изменить следующим образом:

```
{ $N+, E+ }
type
  RealType = real;
var
  epsilon, eps1 : RealType;
begin
  epsilon := 1;
  repeat
    epsilon := epsilon/2;
    eps1 := 1 + epsilon
  until eps1 = 1;
  WriteLn(2*epsilon);
end.
```

Следует учесть, что тип *REAL* оптимизирован для работы без сопро-цессора. Если Ваш ПК оснащен сопроцессором, использование типа *REAL* приведет к дополнительным затратам времени на преобразование *REAL* к *EXTENDED*. Поэтому никогда не используйте REAL на ПК с сопроцессором, т.к. дополнительные затраты времени на преобразование типов могут

свести на нет все преимущества сопроцессора. При разработке программ, критичных ко времени счета, следует заменять его типами *SINGLE* или *DOUBLE*: по сравнению с типом *REAL* скорость вычислений на машинах с сопроцессором в этом случае увеличивается в 2...3 раза. Если в ПК нет арифметического сопроцессора, скорость обработки данных всех вещественных типов приблизительно одинакова.

Особое положение в Турбо Паскале занимает тип *COMP*, который трактуется как вещественное число без экспоненциальной и дробной частей. Фактически, *COMP* — это «большое» целое число со знаком, сохраняющее 19...20 значащих десятичных цифр (во внутреннем представлении *COMP* занимает 8 смежных байт). В то же время в выражениях *COMP* полностью совместим с любыми другими вещественными типами: над ним определены все вещественные операции, он может использоваться как аргумент математических функций и т.д. Наиболее подходящей областью применения типа *COMP* являются бухгалтерские расчеты: денежные суммы выражаются в копейках или центах и действия над ними сводятся к операциям с достаточно длинными целыми числами.

Для работы с вещественными данными могут использоваться встроенные математические функции, представленные в табл. 2.5. В этой таблице *REAL* означает любой вещественный тип, *INTEGER* — любой целый тип.

Таблица 2.5

Стандартные математические функции Турбо Паскаля

Обращение	Тип параметра	Тип результата	Примечание
<code>abs(x)</code>	<code>real</code> , <code>integer</code>	Тип аргумента	Модуль аргумента
<code>ArcTan(x)</code>	<code>real</code>	<code>real</code>	Арктангенс (значение в радианах)
<code>cos(x)</code>	То же	То же	Косинус, угол в радианах
<code>exp(x)</code>	"	"	Экспонента
<code>frac(x)</code>	"	"	Дробная часть числа
<code>int(x)</code>	"	"	Целая часть числа
<code>ln(x)</code>	"	"	Логарифм натуральный
<code>pi</code>	-	"	$\pi = 3.141592653...$
<code>Random</code>	-	"	Псевдослучайное число, равномерно распределенное в диапазоне 0...[1]
<code>Random(x)</code>	<code>integer</code>	<code>integer</code>	Псевдослучайное целое число, равномерно распределенное в диапазоне 0...(x-1)
<code>Randomize</code>	-	-	Инициация генератора псевдослучайных чисел
<code>sin(x)</code>	<code>real</code>	<code>real</code>	Синус, угол в радианах
<code>sqr(x)</code>	То же	То же	Квадрат аргумента
<code>sqrt(x)</code>	"	"	Корень квадратный

4.2. СТРУКТУРИРОВАННЫЕ ТИПЫ

Любой из структурированных типов (а в Турбо Паскале их четыре: массивы, записи, множества и файлы) характеризуется множественностью образующих этот тип элементов, т.е. переменная или константа структурированного типа всегда имеет несколько компонентов. Каждый компонент, в свою очередь, может принадлежать структурированному типу, что позволяет говорить о возможной вложенности типов. В Турбо Паскале допускается произвольная глубина вложенности типов, однако суммарная длина любого из них во внутреннем представлении не должна превышать 65520 байт.

В целях совместимости со стандартным Паскалем в Турбо Паскале разрешается перед описанием структурированного типа ставить зарезервированное слово **PACKED**, предписывающее компилятору, по возможности, экономить память, отводимую под объекты структурированного типа; но компилятор фактически игнорирует это указание: «упаковка» данных в Турбо Паскале осуществляется автоматически везде, где это возможно.

4.2.1. Массивы

Массивы в Турбо Паскале во многом схожи с аналогичными типами данных в других языках программирования. Отличительная особенность массивов заключается в том, что все их компоненты суть данные одного типа (возможно, структурированного). Эти компоненты можно легко упорядочить и обеспечить доступ к любому из них простым указанием его порядкового номера, например:

```

type
  digit = array [0..9] of char;
  matrix = array [byte] of single;
var
  m : matrix;
  d : digit;
  i : integer;
begin
  .....
  m[17] := ord(d[i-1])/10;
  .....
end.
```

Описание типа массива задается следующим образом:

<имя типа> = ARRAY [<сп.инд.типов>] OF <тип>

Здесь **<имя типа>** — правильный идентификатор;

ARRAY, OF — зарезервированные слова (*массив, из*);

<сп.инд.типов> — список из одного или нескольких индексных типов, разделенных запятыми; квадратные скобки, обрамляющие список, — требование синтаксиса;

<тип> — любой тип Турбо Паскаля.

В качестве индексных типов в Турбо Паскале можно использовать любые порядковые типы, кроме *LONGINT* и типов — диапазонов с базовым типом *LONGINT*.

Определить переменную как массив можно и непосредственно при описании этой переменной, без предварительного описания типа массива, например:

```
var
  a,b : array [1..10] of real;
```

Обычно в качестве индексного типа используется тип — диапазон, в котором задаются границы изменения индексов. Так как тип *<min>*, идущий за словом **OF**, — любой тип Турбо Паскаля, то он может быть, в частности, и другим массивом, например:

```
type
  mat = array [0..5] of array [-2..2] of array [char] of byte;
```

Такую запись можно заменить более компактной:

```
type
  mat = array [0..5,-2..2,char] of byte;
```

Глубина вложенности структурированных типов вообще, а следовательно, и массивов — произвольная, поэтому количество элементов в списке индексных типов (размерность массива) не ограничено, однако суммарная длина внутреннего представления любого массива, как уже говорилось, не может быть больше 65520 байт. В памяти ПК элементы массива следуют друг за другом так, что при переходе от младших адресов к старшим наиболее быстро меняется самый правый индекс массива. Если, например,

```
var
  a : array [1..2,1..2] of byte;
begin
  a[1,1]:=1;
  a[2,1]:=2;
  a[1,2]:=3;
  a[2,2]:=4;
end.
```

то в памяти последовательно друг за другом будут расположены байты со значениями 1, 3, 2, 4. Это обстоятельство может оказаться важным при использовании стандартной процедуры копирования памяти *MOVE*.

В Турбо Паскале можно одним оператором присваивания передать все элементы одного массива другому массиву того же типа, например:

```
var
  a,b : array [1..5] of single;
begin
  .....
  a := b;
```

```
.....
end.
```

После этого присваивания все пять элементов массива *A* получают те же значения, что и в массиве *B*. Однако над массивами не определены операции отношения. Нельзя, например, записать

```
if a = b then ...
```

Сравнить два массива можно поэлементно, например:

```
var
  a,b : array [1..5] of single;
  eq   : Boolean;
  i    : byte;
begin
  ....
  eq := true;
  for i := 1 to 5 do
    if a[i] <> b[i] then
      eq := false;
  if eq then
    ....
end.
```

4.2.2. Записи

Запись — это структура данных, состоящая из фиксированного числа компонентов, называемых полями записи. В отличие от массива, компоненты (поля) записи могут быть различного типа. Чтобы можно было ссылаться на тот или иной компонент записи, поля именуются.

Структура объявления типа записи такова:

```
<имя типа> = RECORD <сп.полей> END
```

Здесь <имя типа> — правильный идентификатор;

RECORD, **END** — зарезервированные слова (*запись, конец*);

<сп.полей> — список полей; представляет собой последовательность *разделов записи*, между которыми ставится точка с запятой.

Каждый раздел записи состоит из одного или нескольких идентификаторов полей, отделяемых друг от друга запятыми. За идентификатором (идентификаторами) ставится двоеточие и описание типа поля (полей), например:

```
type
  BirthDay = record
    day, month : byte;
    year       : word
  end;
```

```
var
  a,b : Birthday;
  .....
```

В этом примере тип *BIRTHDAY* (*день рождения*) есть запись с полями *DAY*, *MONTH* и *YEAR* (*день, месяц и год*); переменные *A* и *B* содержат записи типа *BIRTHDAY*.

Как и в массиве, значения переменных типа записи можно присваивать другим переменным того же типа, например

```
a := b;
```

К каждому из компонентов записи можно получить доступ, если использовать составное имя, т.е. указать имя переменной, затем точку и имя поля:

```
a.day := 27;
b.year := 1939;
```

Для вложенных полей приходится продолжать уточнения:

```
type
  Birthday = record
    day, month: byte;
    year      : word
  end;
var
  c : record
    name : string;
    bd   : Birthday
  end;
begin
  .....
  if c.bd.year = 1939 then ...
end.
```

Чтобы упростить доступ к полям записи, используется оператор присоединения *WITH*:

```
WITH <переменная> DO <оператор>
```

Здесь *WITH*, *DO* — ключевые слова (*с, делать*);

<переменная> — имя переменной типа записи, за которым, возможно, следует список вложенных полей;

<оператор> — любой оператор Турбо Паскаля.

Например:

```
with c.bd do month := 9;
```

Это эквивалентно

```
with c do with bd do month := 9;
```

или

```
with c, bd do month := 9;
```

или

```
c.bd.month := 9;
```

Турбо Паскаль разрешает использовать записи с так называемыми вариантными полями, например:

```
type
  Forma = record
    Name: string;
    case byte of
      0: (BirthPlace: string [40]);
      1: (Country   : string [20];
         EntryPort : string [20];
         EntryDate : 1..31;
         ExitDate  : 1..31)
    end;
```

В этом примере тип *FORMA* определяет запись с одним фиксированным полем *NAME* и вариантной частью, которая задается предложением *CASE... OF*. Вариантная часть состоит из нескольких вариантов (в примере — из двух вариантов: 0 и 1). Каждый вариант определяется константой выбора, за которой следует двоеточие и список полей, заключенный в круглые скобки. В любой записи может быть только одна вариантная часть, и, если она есть, она должна располагаться за всеми фиксированными полями.

Замечательной особенностью вариантной части является то обстоятельство, что все заданные в ней варианты «накладываются» друг на друга, т.е. каждому из них выделяется одна и та же область памяти. Это открывает дополнительные возможности преобразования типов, например:

```
var
  mem4 : record
    case byte of
      0 : (by : array [0..3] of byte);
      1 : (wo : array [0..1] of word);
      2 : (lo : longint);
    end;
```

В этом примере запись *MEM4* имеет три варианта, каждый из которых занимает в памяти один и тот же участок из 4 байт. В зависимости от того, к какому полю записи мы обращаемся в программе, этот участок может рассматриваться как массив из 4 байт (поле *BY*), массив из двух целых типа *WORD* (поле *WO*) или, наконец, как одно целое число типа *LONGINT* (поле *LO*). Например, этой записи можно сначала присвоить значение как длинному целому, а затем проанализировать результат по **байтам** или **словам**:

```
var
  x : word;
```

```

xb: byte;
xl: LongInt;
begin
    .....
    with m do
        begin
            lo := trunc(2*pi*x);
            if wo[1] = 0 then
                if by[1] = 0 then
                    xb := x[0]
                else
                    x := wo[0]
                else
                    xl := lo
            end;
        .....
    end.

```

Предложение *CASE... OF*, открывающее вариантную часть, внешне похоже на соответствующий оператор выбора, но на самом деле лишь играет роль своеобразного служебного слова, обозначающего начало вариантной части. Именно поэтому в конце вариантной части не следует ставить *END* как пару к *CASE... OF*. (Поскольку вариантная часть — всегда последняя в записи, за ней все же стоит *END*, но лишь как пара к *RECORD*). Ключ выбора в предложении *CASE... OF* фактически игнорируется компилятором: единственное требование, предъявляемое к нему Турбо Паскалем, состоит в том, чтобы ключ определял некоторый стандартный или предварительно объявленный порядковый тип. Причем сам этот тип никак не влияет ни на количество следующих ниже вариантных полей, ни даже на характер констант выбора. В стандартном Паскале в качестве ключа выбора необходимо указывать некоторую переменную порядкового типа, причем в исполняемой части программы можно присваивать значение этой переменной и таким образом влиять на выбор полей. В Турбо Паскале также можно в поле ключа выбора указывать переменную порядкового типа и даже присваивать ей в программе значение, что однако не влияет на выбор поля: значения констант выбора в Турбо Паскале могут быть произвольными, в том числе повторяющимися, например:

```

type
    rec1 = record
        a : byte;
        b : word;
    end;
    rec2 = record
        c : longint;
        case x : byte of
            1 : (d : word);
            2 : (e : record

```

```

        case Boolean of
          3 : ( f : rec1);
          3 : ( g : single);
          '3' : ( c : word)
        end)
    end;
var
  r : rec2;
begin
  r.x := 255;
  if r.e.g = 0 then
    WriteLn('O.K.')
  else
    WriteLn(r.e.g)
  end.
end.

```

В этом примере предложение

```
case Boolean of
```

в записи, определяемой в поле *E*, объявляет ключом выбора логический тип, который, как известно, имеет лишь два значения — *TRUE* и *FALSE*. Константы же выбора следующих далее вариантов не только содержат совершенно не свойственные этому типу значения, но и две из них повторяются, а общее количество вариантов — три, а не два, как следовало бы ожидать.

Имена полей должны быть уникальными в пределах той записи, где они объявлены, однако, если записи содержат поля — записи, т.е. вложены одна в другую, имена могут повторяться на разных уровнях вложенности (см. поле *C* в последнем примере).

4.2.3. Множества

Множества — это наборы однотипных логически связанных друг с другом объектов. Характер связей между объектами лишь подразумевается программистом и никак не контролируется Турбо Паскалем. Количество элементов, входящих в множество, может меняться в пределах от 0 до 256 (множество, не содержащее элементов, называется *пустым*). Именно непостоянством количества своих элементов множества отличаются от массивов и записей.

Два множества считаются эквивалентными тогда и только тогда, когда все их элементы одинаковы, причем порядок следования элементов в множестве безразличен. Если все элементы одного множества входят также и в другое, говорят о включении первого множества во второе. Пустое множество включается в любое другое.

Пример определения и задания множеств:

```

type
  digitChar = set of '0'..'9';
  digit      = set of 0..9;

```



```

var
  s1,s2,s3 : digitChar;
  s4,s5,s6 : digit;
begin
  .....
  s1 := ['1', '2', '3'];
  s2 := ['3', '2', '1'];
  s3 := ['2', '3'];
  s4 := [0..3, 6];
  s5 := [4, 5];
  s6 := [3..9];
  .....
end.

```

В этом примере множества *S1* и *S2* эквивалентны, а множество *S3* включено в *S2*, но не эквивалентно ему.

Описание типа множества имеет вид:

<имя типа> = **SET OF** <баз.тип>

Здесь <имя типа> — правильный идентификатор;

SET, OF — зарезервированные слова (множество, из);

<баз.тип> — базовый тип элементов множества, в качестве которого может использоваться любой порядковый тип, кроме *WORD, INTEGER, LONGINT*.

Для задания множества используется так называемый *конструктор множества*: список спецификаций элементов множества, отделяемых друг от друга запятыми; список обрамляется квадратными скобками (см. предыдущий пример). Спецификациями элементов могут быть константы или выражения базового типа, а также — тип-диапазон того же базового типа.

Над множествами определены следующие операции:

- * пересечение множеств; результат содержит элементы, общие для обоих множеств; например, *S4*S6* содержит [3], *S4*S5* — пустое множество (см. выше);
- + объединение множеств; результат содержит элементы первого множества, дополненные недостающими элементами из второго множества:
 $S4+S5$ содержит [0,1,2,3,4,5,6];
 $S5+S6$ содержит [3,4,5,6,7,8,9];
- разность множеств; результат содержит элементы из первого множества, которые не принадлежат второму:
 $S6-S5$ содержит [3,6,7,8,9];
 $S4-S5$ содержит [0,1,2,3,6];
- = проверка эквивалентности; возвращает *TRUE*, если оба множества эквивалентны;

- <> проверка неэквивалентности; возвращает *TRUE*, если оба множества неэквивалентны;
- <= проверка вхождения; возвращает *TRUE*, если первое множество включено во второе;
- >= проверка вхождения; возвращает *TRUE*, если второе множество включено в первое;
- IN* проверка принадлежности; в этой бинарной операции первый элемент — выражение, а второй — множество одного и того же типа; возвращает *TRUE*, если выражение имеет значение, принадлежащее множеству:

```
3 in s6 возвращает TRUE;
2*2 in s1 возвращает FALSE.
```

Дополнительно к этим операциям можно использовать две процедуры.

INCLUDE — включает новый элемент во множество. Обращение к процедуре:

```
INCLUDE (S, I)
```

Здесь *S* — множество, состоящее из элементов базового типа *TSetBase*;

I — элемент типа *TSetBase*, который необходимо включить во множество.

EXCLUDE — исключает элемент из множества. Обращение:

```
EXCLUDE (S, I)
```

Параметры обращения — такие же, как у процедуры *INCLUDE*.

В отличие от операций $+$ и $-$, реализующих аналогичные действия над двумя множествами, процедуры оптимизированы для работы с одиночными элементами множества и поэтому отличаются высокой скоростью выполнения.

В примере 4.1, иллюстрирующем приемы работы с множествами, реализуется алгоритм выделения из первой сотни натуральных чисел всех простых чисел. В его основе лежит прием, известный под названием «решето Эратосфена». В соответствии с этим алгоритмом вначале формируется множество *BEGINSET*, состоящее из всех целых чисел в диапазоне от 2 до *N*. В множество *PRIMERSET* (оно будет содержать искомые простые числа) помещается 1. Затем циклически повторяются следующие действия:

- взять из *BEGINSET* первое входящее в него число *NEXT* и поместить его в *PRIMERSET*;
- удалить из *BEGINSET* число *NEXT* и все другие числа, кратные ему, т.е. $2 \cdot \text{NEXT}$, $3 \cdot \text{NEXT}$ и т.д.

Цикл повторяется до тех пор, пока множество *BEGINSET* не станет пустым.

Эту программу нельзя использовать для произвольного *N*, так как в любом множестве не может быть больше 256 элементов.

Пример 4.1

```

Program Primer_numbers_detect;
{Выделение всех простых чисел из первых N целых}
const
  N = 255; {Количество элементов исходного множества}
type
  SetOfNumber = set of 1..N;
var
  n1,next,i : word;          {Вспомогательные переменные}
  BeginSet,   {Исходное множество}
  PrimerSet : SetOfNumber;   {Множество простых чисел}
begin
  BeginSet := [2..N];        {Создаем исходное множество}
  PrimerSet:= [1];           {Первое простое число}
  next      := 2;            {Следующее простое число}
  while BeginSet <> [] do    {Начало основного цикла}
    begin
      n1 := next; {n1-число, кратное очередному простому (next)}
      {Цикл удаления из исходного множества непростых чисел:}
      while n1 <= N do
        begin
          Exclude(BeginSet,n1);
          n1 := n1+next      {Следующее кратное}
        end; {Конец цикла удаления}
      Include(PrimerSet,next);
      {Получаем следующее простое, которое есть первое
      невычеркнутое из исходного множества}
      repeat
        inc(next)
      until (next in BeginSet) or (next > N)
    end; {Конец основного цикла}
    {Выводим результат:}
    for i := 1 to N do
      if i in PrimerSet then Write(i:8);
    WriteLn
  END.

```

Перед тем как закончить рассмотрение множеств полезно провести небольшой эксперимент. Измените описание типа *SETOFNUMBER* следующим образом:

```

type
  SetOfNumber = set of 1..1;

```

и еще раз запустите программу из предыдущего примера. На экран будет выведено

1 3 5 7

Множества `BeginSet` и `PrimerSet` состоят теперь из одного элемента, а программа сумела поместить в них не менее семи!

Секрет этого прост: внутреннее устройство множества таково, что каждому его элементу ставится в соответствие один двоичный разряд (один бит); если элемент включен во множество, соответствующий разряд имеет значение 1, в противном случае — 0. Минимальной единицей памяти является один байт, содержащий 8 бит. Компилятор выделил множествам по одному байту, в результате мощность каждого из них стала равна 8 элементов. Максимальная мощность множества — 256 элементов. Для таких множеств компилятор выделяет по 16 смежных байт.

И еще один эксперимент: измените диапазон базового типа на 1..256. Хотя мощность этого типа составляет 256 элементов, при попытке компиляции программы компилятор сообщает:

Error 23: Set base type out of range.

(Ошибка 23: Базовый тип множества выходит за допустимые границы.)

Компилятор разрешает использовать в качестве базового типа целочисленный тип — диапазон с минимальной границей 0 и максимальной 255 или любой перечисляемый тип не более чем с 256 элементами (максимальная мощность перечисляемого типа — 65536 элементов).

4.3. СТРОКИ

Тип **STRING** (строка) в Турбо Паскале широко используется для обработки текстов. Он во многом похож на одномерный массив символов **ARRAY [0..N] OF CHAR**, однако, в отличие от последнего, количество символов в строке — переменной может меняться от 0 до N , где N — максимальное количество символов в строке. Значение N определяется объявлением типа **STRING [N]** и может быть любой константой порядкового типа, но не больше 255. Турбо Паскаль разрешает не указывать N , в этом случае длина строки принимается максимально возможной, а именно $N=255$.

Строка в Турбо Паскале трактуется как цепочка символов. К любому символу в строке можно обратиться точно так же, как к элементу одномерного массива **ARRAY [0..N] OF CHAR**, например:

```
var st : string;
begin
    ....
    if st[5] = 'A' then ...
end.
```

Самый первый байт в строке имеет индекс 0 и содержит текущую длину строки. Первый значащий символ строки занимает второй байт и имеет индекс 1. Над длиной строки можно осуществлять необходимые действия и таким способом изменять длину. Например, удалить из строки все ведомые пробелы можно следующим образом:

```
var
```

```

    st : string;
    i : Byte;
begin
    .....
    i := ord(st[0]);      {i - текущая длина строки}
    while (i <> 0) and (st[i] = ' ') do
        begin
            dec(i);
            st[0] := chr(i)
        end;
    .....
end.

```

Значение $ORD(st[0])$, т.е. текущую длину строки, можно получить и с помощью функции $LENGTH(st)$, например:

```

while (Length(st)<>0) and (st[Length(st)]=' ') do
    st[0] := chr(Length(st)-1)

```

К строкам можно применять операцию «+» — сцепление, например:

```

    st := 'a' + 'b';
    st := st + 'c'; {st содержит "abc"}

```

Если длина сцепленной строки превысит максимально допустимую длину N , то «лишние» символы отбрасываются. Следующая программа, например, напечатает символ 1:

```

var
    st: string [1];
begin
    st:='123';
    WriteLn(st)
end.

```

Все остальные действия над строками и символами реализуются с помощью описываемых ниже стандартных процедур и функций.

CONCAT(S1 [,S2, ...,SN]) — функция типа **STRING**; возвращает строку, представляющую собой сцепление строк-параметров $S1, S2, \dots, SN$.

COPY(ST, INDEX, COUNT) — функция типа **STRING**; копирует из строки ST $COUNT$ символов, начиная с символа с номером $INDEX$.

DELETE(ST, INDEX, COUNT) — процедура; удаляет $COUNT$ символов из строки ST , начиная с символа с номером $INDEX$.

INSERT(SUBST, ST, INDEX) — процедура; вставляет подстроку $SUBST$ в строку ST , начиная с символа с номером $INDEX$.

LENGTH(ST) — функция типа **INTEGER**; возвращает длину строки ST .

POS(SUBST, ST) — функция типа *INTEGER*; отыскивает в строке *ST* первое вхождение подстроки *SUBST* и возвращает номер позиции, с которой она начинается; если подстрока не найдена, возвращается ноль.

STR(X [:WIDTH [:DECIMALS]], ST) — процедура; преобразует число *X* любого вещественного или целого типов в строку символов *ST* так, как это делает процедура *WRITELN* перед выводом; параметры *WIDTH* и *DECIMALS*, если они присутствуют, задают формат преобразования: *WIDTH* определяет общую ширину поля, выделенного под соответствующее символьное представление вещественного или целого числа *X*, а *DECIMALS* — количество символов в дробной части (этот параметр имеет смысл только в том случае, когда *X* — вещественное число).

VAL(ST, X, CODE) — процедура; преобразует строку символов *ST* во внутреннее представление целой или вещественной переменной *X*, которое определяется типом этой переменной; параметр *CODE* содержит ноль, если преобразование прошло успешно, и тогда в *X* помещается результат преобразования, в противном случае он содержит номер позиции в строке *ST*, где обнаружен ошибочный символ, и в этом случае содержимое *X* не меняется; в строке *ST* могут быть ведущие пробелы, однако ведомые пробелы недопустимы; например, обращение `val(' 123', k, i)` пройдет успешно: *k* получит значений 123, в *i* будет помещен 0, в то время как обращение `val('123 ', k, i)` будет ошибочным: значение *k* не изменится, а *i* будет содержать 4.

UPCASE(CH) — функция типа *CHAR*; возвращает для символьного выражения *CH*, которое должно представлять собой строчную латинскую букву, соответствующую заглавную букву; если значением *CH* является любой другой символ (в том числе строчная буква русского алфавита), функция возвращает его без преобразования.

Примеры:

```
var
  x : real;
  y : integer;
  st, st1: string;
begin
  st := concat('12', '345');           {строка st содержит 12345}
  st1 := copy(st, 3, Length(st)-2);    {st1 содержит 345}
  insert('-', st1, 2);                 {строка st1 содержит 3-45}
  delete(st, pos('2', st), 3);         {строка st содержит 15}
  str(pi:6:2, st);                     {строка st содержит 3.14}
  val('3,1415', x, y);                 {y содержит 2, x остался
                                         без изменения}
end.
```

Операции отношения `=`, `<>`, `>`, `<`, `>=`, `<=` выполняются над двумя строками посимвольно, слева направо с учетом внутренней кодировки символов (см. табл.4.1 и прил.2). Если одна строка меньше другой по дли-

не, недостающие символы короткой строки заменяются значением *CHR(0)*

Следующие операции отношения дадут значение TRUE:

```
' ' < ' .'  
'A' > '1'  
'Turbo' < 'Turbo Pascal'  
'Паскаль' > 'Turbo Pascal'
```

4.4. СОВМЕСТИМОСТЬ И ПРЕОБРАЗОВАНИЕ ТИПОВ

Как уже неоднократно отмечалось, Турбо Паскаль — это типизированный язык. Он построен на основе строгого соблюдения концепции типов, в соответствии с которой все применяемые в языке операции определены только над операндами совместимых типов. При обсуждении операций над вещественными данными мы уже затрагивали проблему совместимости вещественных и целых типов. Аналогичные проблемы возникают при операциях над строками разной длины, строками и символами и т.д. Ниже приводится более полное определение совместимости типов.

Два типа считаются совместимыми, если:

- оба они есть один и тот же тип;
- оба вещественные;
- оба целые;
- один тип есть тип — диапазон второго типа;
- оба являются типами — диапазонами одного и того же базового типа;
- оба являются множествами, составленными из элементов одного и того же базового типа;
- оба являются упакованными строками (определены с предшествующим словом *PACKED*) одинаковой максимальной длины;
- один тип есть тип — строка, а другой — тип — строка, упакованная строка или символ;
- один тип есть любой указатель, а другой — нетипизированный указатель;
- один тип есть указатель на объект, а другой — указатель на родственный ему объект;
- оба есть процедурные типы с одинаковыми типом результата (для типа — функции), количеством параметров и типом взаимно соответствующих параметров.

Совместимость типов приобретает особое значение в операторах присваивания. Пусть *T1* — тип переменной, а *T2* — тип выражения, т.е. выполняется присваивание *T1 := T2*. Это присваивание возможно в следующих случаях:

- *T1* и *T2* есть один и тот же тип и этот тип не относится к файлам или массивам файлов, или записям, содержащим поля — файлы, или массивам таких записей;

- $T1$ и $T2$ являются совместимыми порядковыми типами и значение $T2$ лежит в диапазоне возможных значений $T1$;
- $T1$ и $T2$ являются вещественными типами и значение $T2$ лежит в диапазоне возможных значений $T1$;
- $T1$ — вещественный тип и $T2$ — целый тип;
- $T1$ — строка и $T2$ — символ;
- $T1$ — строка и $T2$ — упакованная строка;
- $T1$ и $T2$ — совместимые упакованные строки;
- $T1$ и $T2$ — совместимые множества и все члены $T2$ принадлежат множеству возможных значений $T1$;
- $T1$ и $T2$ — совместимые указатели;
- $T1$ и $T2$ — совместимые процедурные типы;
- $T1$ — объект и $T2$ — его потомок.

В программе данные одного типа могут преобразовываться в данные другого типа. Такое преобразование может быть явным или неявным.

При *явном преобразовании* типов используются вызовы специальных функций преобразования, аргументы которых принадлежат одному типу, а значение — другому. Таковыми являются уже рассмотренные функции *ORD*, *TRUNC*, *ROUND*, *CHR*. В гл. 6 описывается функция *PTR*, преобразующая четырехбайтный целочисленный аргумент к типу — указателю.

В Турбо Паскале может использоваться и более общий механизм преобразования типов, согласно которому преобразование достигается применением идентификатора (имени) стандартного типа или типа, определенного пользователем, как идентификатора функции преобразования к выражению преобразуемого типа (так называемое автоопределенное преобразование типов). Например, допустимы следующие вызовы функций:

```

type
  MyType = (a, b, c, d);
.....
MyType (2)
integer ('D')
pointer (longint(a) + $FF)
char (127 mod c)
byte (k)

```

При автоопределенном преобразовании типа выражения может произойти изменение длины его внутреннего представления (длина может увеличиться или уменьшиться).

В Турбо Паскале определен еще один явный способ преобразования данных: в ту область памяти, которую занимает переменная некоторого типа, можно поместить значение выражения другого типа, если только длина внутреннего представления вновь размещаемого значения в точности равна длине внутреннего представления переменной. С этой целью вновь используется автоопределенная функция преобразования типов, но уже в левой части оператора присваивания:


```

type
  byt = array [1..2] of byte;
  int = array [1..2] of integer;
  rec = record
    x, y : integer
  end;

var
  vbyt : byt;
  vint : int;
  vrec : rec;

begin
  byt(vint[1])[2] := 0;
  int(vrec[1]) := 256
end.

```

Неявное преобразование типов возможно только в двух случаях:

- в выражениях, составленных из вещественных и целочисленных переменных, последние автоматически преобразуются к вещественному типу, и все выражение в целом приобретает вещественный тип;
- одна и та же область памяти попеременно трактуется как содержащая данные то одного, то другого типа (совмещение в памяти данных разного типа).

Совмещение данных в памяти может произойти при использовании записей с вариантными полями (см. 4.2.2), типизированных указателей, содержащих одинаковый адрес (см. гл. 6), а также при явном размещении данных разного типа по одному и тому же абсолютному адресу. Для размещения переменной по нужному абсолютному адресу она описывается с последующей стандартной директивой *ABSOLUTE*, за которой помещается либо абсолютный адрес, либо идентификатор ранее определенной переменной. Абсолютный адрес указывается парой чисел типа *WORD*, разделенных двоеточием; первое число трактуется как сегмент, второе — как смещение адреса (см. гл. 6). Например:

```

b : byte absolute $0000:$0055;
w : LongInt absolute 128:0;

```

Если за словом *ABSOLUTE* указан идентификатор ранее определенной переменной, то происходит совмещение в памяти данных разного типа, причем первые байты внутреннего представления этих данных будут располагаться по одному и тому же абсолютному адресу, например:

```

var
  x : real;
  y : array [1..3] of integer absolute x;

```

В этом примере переменные *X* и *Y* будут размещены, начиная с одного и того же абсолютного адреса. Таким образом, одну и ту же область памяти длиной 6 байт, а следовательно, и размещенные в этой области данные теперь можно рассматривать как данные либо типа *REAL*, либо как

массив из трех данных типа *INTEGER*. Например, следующая программа выдаст на экран содержимое первых двух байт внутреннего представления вещественного числа $\pi = 3.1415$ в виде целого числа:

```
var
  x : real;
  y : array [1..3] of integer absolute x;
begin
  x := pi;
  WriteLn(y[1])
end.
```

На экран будет выдан результат 8578.

Неявные преобразования типов могут служить источником труднообнаруживаемых ошибок в программе, поэтому везде, где это возможно, следует избегать их.

Глава 5

ФАЙЛЫ

Под *файлом* понимается либо именованная область внешней памяти ПК (жесткого диска, гибкой дискеты, электронного «виртуального» диска), либо логическое устройство — потенциальный источник или приемник информации.

Любой файл имеет три характерные особенности. Во-первых, у него есть имя, что дает возможность программе работать одновременно с несколькими файлами. Во-вторых, он содержит компоненты одного типа. Типом компонентов может быть любой тип Турбо Паскаля, кроме файлов. Иными словами, нельзя создать «файл файлов». В-третьих, длина вновь создаваемого файла никак не оговаривается при его объявлении и ограничивается только емкостью устройств внешней памяти.

Файловый тип или переменную файлового типа можно задать одним из трех способов:

```
<имя> = FILE OF <тип>;  
      <имя> = TEXT;  
      <имя> = FILE;
```

Здесь <имя> — имя файлового типа (правильный идентификатор);

FILE, **OF** — зарезервированные слова (файл, из);

TEXT — имя стандартного типа текстовых файлов;

<тип> — любой тип Турбо Паскаля, кроме файлов.

Например:

```
type  
  product = record  
    name : string;  
    code : word;  
    cost : comp  
  end;  
  text80 = file of string [80];  
var  
  f1 : file of char;  
  f2 : text;  
  f3 : file;  
  f4 : text80;  
  f5 : file of product;
```

В зависимости от способа объявления можно выделить три вида файлов:

- типизированные файлы (задаются предложением **FILE OF...**);
- текстовые файлы (определяются типом **TEXT**);
- нетипизированные файлы (определяются типом **FILE**).

В наших примерах *F1*, *F4* и *F5* — типизированные файлы, *F2* — текстовый файл, *F3* — нетипизированный файл. Вид файла, вообще говоря, определяет способ хранения информации в файле. Однако в Турбо Паскале нет средств контроля вида ранее созданных файлов. При объявлении уже существующих файлов программист должен сам следить за соответствием вида объявления характеру файла.

5.1. ДОСТУП К ФАЙЛАМ

Любой программе доступны два предварительно объявленных файла со стандартными файловыми переменными: *INPUT* — для чтения данных с клавиатуры и *OUTPUT* — для вывода на экран. Стандартный Паскаль требует обязательного упоминания этих файлов в заголовке программы, например, так:

```
PROGRAM NameOfProgram(input,output);
```

В Турбо Паскале это необязательно, вот почему заголовок программы можно опускать.

Любые другие файлы, а также логические устройства становятся доступны программе только после выполнения особой процедуры открытия файла (логического устройства). Эта процедура заключается в связывании ранее объявленной файловой переменной с именем существующего или вновь создаваемого файла, а также в указании направления обмена информацией: чтение из файла или запись в него.

Файловая переменная связывается с именем файла в результате обращения к стандартной процедуре *ASSIGN* :

```
ASSIGN (<ф.п.>, <имя файла или л.у.>);
```

Здесь <ф.п.> — файловая переменная (правильный идентификатор, объявленный в программе как переменная файлового типа);

<имя файла или л.у.> — текстовое выражение, содержащее имя файла или логическое устройство.

Если имя файла задается в виде пустой строки, например, *ASSIGN(f,")*, то в зависимости от направления обмена данными файловая переменная связывается со стандартным файлом *INPUT* или *OUTPUT*.

5.1.1. Имена файлов

Имя файла — это любое выражение строкового типа, которое строится по правилам определения имен в MS-DOS (операционной системе ПК):

- имя содержит до восьми разрешенных символов; разрешенные символы — это прописные и строчные латинские буквы, цифры и символы:

! @ # \$ % ^ & () ' ~ - _

- имя начинается с любого разрешенного символа;

- за именем может следовать расширение — последовательность до трех разрешенных символов; расширение, если оно есть, отделяется от имени точкой.

Перед именем может указываться так называемый путь к файлу: имя диска и/или имя текущего каталога и имена каталогов вышестоящих уровней.

Имя диска — это один из символов *A...Z*, после которого ставится двоеточие. Имена *A:* и *B:* относятся к дисковым накопителям на гибких дискетах, имена *C:*, *D:* и т.д. — к жестким дискам. Эти имена могут относиться также к одному или нескольким виртуальным дискам, созданным в оперативной памяти ПК специальной командой *VDISK* в ходе выполнения файла автоустановки *CONFIG.SYS* дисковой операционной системы.

Если имя диска не указано, подразумевается устройство по умолчанию — то, которое было установлено в операционной системе перед началом работы программы.

За именем диска может указываться имя каталога, содержащего файл. Если имени каталога предшествует обратная косая черта, то путь к файлу начинается из корневого каталога, если черты нет — из текущего каталога, установленного в системе по умолчанию. За именем каталога может следовать одно или несколько имен каталогов нижнего уровня. Каждому из них должна предшествовать обратная косая черта. Весь путь к файлу отделяется от имени файла обратной косой чертой. Максимальная длина имени вместе с путем — 79 символов, например:

```
var
    finp : text;
    fout : file of string;
const
    name = 'c:\dir\subdir\out.txt';
begin
    assign(finp, '123.dat');
    assign(fout, name);
end.
```

5.1.2. Логические устройства

Стандартные аппаратные средства ПК, такие как клавиатура, экран дисплея, печатающее устройство (принтер) и коммуникационные каналы ввода—вывода, определяются в Турбо Паскале специальными именами, которые называются *логическими устройствами*. Все они в Турбо Паскале рассматриваются как потенциальные источники или приемники текстовой информации.

CON — логическое имя, которое определяет консоль — клавиатуру или экран дисплея. Турбо Паскаль устанавливает различие между этими физическими устройствами по направлению передачи данных: чтение данных возможно только с клавиатуры, а запись — только на экран. Таким образом, с помощью логического устройства *CON* нельзя, например, прочитать данные с экрана ПК, хотя такая аппаратная возможность существует.

Ввод с клавиатуры буферируется: символы по мере нажатия на клавиши помещаются в специальный строковый буфер, который передается по — грамме только после нажатия на клавишу *Enter*. Буферизация ввода обеспечивает возможность редактирования вводимой строки стандартными средствами ДОС. При вводе символов осуществляется их эхо — повтор на экране ПК. В Турбо Паскале можно прочитать любой символ клавиатуры, в том числе и символ CR, вырабатываемый клавишей *Enter*, сразу после нажатия на соответствующую клавишу без эхо — повтора.

PRN — логическое имя принтера. Если к ПК подключено несколько принтеров, доступ к ним осуществляется по логическим именам *LPT1*, *LPT2* и *LPT3*. Имена *PRN* и *LPT1* первоначально — синонимы. Средствами ДОС можно присвоить имя *PRN* любому другому логическому устройству, способному принимать информацию.

Стандартный библиотечный модуль *PRINTER*, входящий в библиотеку *TURBO.TPL*, объявляет имя файловой переменной *LST* и связывает его с логическим устройством *LPT1*. Это дает возможность использовать простое обращение к принтеру. Например, программа

```
Uses Printer;  
begin  
  WriteLn(LST, 'Привет, мир!')  
end.
```

выведет на принтер фразу «Привет, мир!», а все необходимые операции по открытию логического устройства выполнит библиотечный блок *PRINTER* (подробности работы с модулями см. в гл. 9).

AUX — логическое имя коммуникационного канала, который обычно используется для связи ПК с другими машинами. Коммуникационный канал может осуществлять и прием, и передачу данных, однако в программе в каждый момент времени ему можно назначить только одну из этих функций. Как правило, в составе ПК имеются два коммуникационных канала, которым даются имена логических устройств *COM1* и *COM2*. Первоначально имена *AUX* и *COM1* — синонимы.

NUL — логическое имя «пустого» устройства. Это устройство чаще всего используется в отладочном режиме и трактуется как устройство — приемник информации неограниченной емкости. При обращении к *NUL* как источнику информации выдается признак конца файла *EOF*.

Связывание логического устройства с файловой переменной осуществляется процедурой *ASSIGN*, например:

```
var  
  fi,fo : text;  
begin  
  assign(fi, 'AUX');  
  assign(fo, 'LPT2');  
end.
```

Турбо Паскаль никогда не связывает имена логических устройств с дисковыми файлами, в этом смысле эти имена можно считать зарезервированными. Иными словами, нельзя, например, обратиться к дисковому

файлу с именем *PRN*, так как Турбо Паскаль всегда интерпретирует такой запрос как обращение к принтеру.

5.1.3. Инициация файла

Инициировать файл означает указать для этого файла направление передачи данных. В Турбо Паскале можно открыть файл для чтения, для записи информации, а также для чтения и записи одновременно.

Для чтения файл иницируется с помощью стандартной процедуры *RESET*:

```
RESET (<ф.п.>);
```

Здесь <ф.п.> — файловая переменная, связанная ранее процедурой *ASSIGN* с уже существующим файлом или логическим устройством — приемником информации.

При выполнении этой процедуры дисковый файл или логическое устройство подготавливается к чтению информации. В результате специальная переменная — указатель, связанная с этим файлом, будет указывать на начало файла, т.е. на компонент с порядковым номером 0.

Если делается попытка инициировать чтение из несуществующего файла или из логического устройства *PRN*, возникает ошибка периода исполнения, которая может быть сообщена программе ненулевым значением встроенной функции *IORESULT* типа *WORD*. Например, следующий фрагмент программы позволяет установить, существует ли требуемый файл на диске:

```
var
  f: file of char;
begin
  assign(f, 'myfile.dat');
  {$I-}      {Отключаем контроль ошибок ввода-вывода}
  reset(f);
  {$I+}      {Включаем контроль ошибок ввода-вывода}
  if IOResult <> 0 then
    ..... {Файл не существует}
  else
    ..... {Файл существует}
end.
```

В этом фрагменте с помощью директивы компилятора *{\$I-}* отключается автоматический контроль ошибок ввода — вывода. Если этого не сделать, то отсутствие файла приведет к аварийному завершению программы.

В Турбо Паскале разрешается обращаться к типизированным файлам, открытым процедурой *RESET* (т.е. для чтения информации), с помощью процедуры *WRITE* (т.е. для записи информации). Такая возможность позволяет легко обновлять ранее созданные типизированные файлы и при необходимости расширять их. Для текстовых файлов, открытых процедурой *RESET*, нельзя использовать процедуру *WRITE* или *WRITELN*.

Стандартная процедура

REWRITE (<ф.п.>)

инициирует запись информации в файл или в логическое устройство, связанное ранее с файловой переменной <ф.п.>. Процедурой *REWRITE* нельзя инициировать запись информации в ранее существовавший дисковый файл: при выполнении этой процедуры старый файл уничтожается и никаких сообщений об этом в программу не передается. Новый файл подготавливается к приему информации и его указатель принимает значение 0.

Стандартная процедура

APPEND (<ф.п.>)

инициирует запись в ранее существовавший текстовый файл для его расширения, при этом указатель файла устанавливается в его конец. Процедура *APPEND* применима только к текстовым файлам, т.е. их файловая переменная должна иметь тип *TEXT* (см. выше). Процедурой *APPEND* нельзя инициировать запись в типизированный или нетипизированный файл. Если текстовый файл ранее уже был открыт с помощью *RESET* или *REWRITE*, использование процедуры *APPEND* приведет к закрытию этого файла и открытию его вновь, но уже для добавления записей.

5.2. ПРОЦЕДУРЫ И ФУНКЦИИ ДЛЯ РАБОТЫ С ФАЙЛАМИ

Ниже описываются процедуры и функции, которые можно использовать с файлами любого вида. Специфика работы с типизированными, текстовыми и нетипизированными файлами рассматривается в следующих разделах.

Процедура *Close*. Закрывает файл, однако связь файловой переменной с именем файла, установленная ранее процедурой *ASSIGN*, сохраняется. Формат обращения:

CLOSE (<ф.п.>)

При создании нового или расширении старого файла процедура обеспечивает сохранение в файле всех новых записей и регистрацию файла в каталоге. Функции процедуры *CLOSE* выполняются автоматически по отношению ко всем открытым файлам при нормальном завершении программы. Поскольку связь файла с файловой переменной сохраняется, файл можно повторно открыть без дополнительного использования процедуры *ASSIGN*.

Процедура *RENAME*. Переименовывает файл. Формат обращения:

RENAME (<ф.п.>, <новое имя>)

Здесь <новое имя> — строковое выражение, содержащее новое имя файла.

Перед выполнением процедуры необходимо закрыть файл, если он ранее был открыт процедурами *RESET*, *REWRITE* или *APPEND*.

Процедура *ERASE*. Уничтожает файл. Формат обращения:

ERASE (<ф.п.>)

Перед выполнением процедуры необходимо закрыть файл, если он ранее был открыт процедурами *RESET*, *REWRITE* или *APPEND*.

Следующий фрагмент программы показывает, как можно использовать процедуры *RENAME* и *CLOSE* при работе с файлами. Предположим, что требуется отредактировать файл, имя которого содержит переменная *NAME*. Перед редактированием необходимо убедиться, что нужный файл имеется на диске, и переименовать его — заменить расширение этого файла на *BAK* (страховочная копия). Если файл с таким расширением уже существует, его надо стереть.

```

var
  fi : text; {Исходный файл}
  fo : text; {Отредактированный файл}
  name      : string;
  name_bak : String;
  k,i: word;

const
  bak = '.bak';

begin
  .....
  {Получаем в name_bak имя файла с расширением .BAK:}
  k := pos('.',name);
  if k = 0 then
    k := length(name) + 1;
  name_bak := copy(name,1,k-1) + bak;
  {Проверяем существование исходного файла:}
  assign(fi,name);
  {$I-}
  reset(fi);
  if IOResult <> 0 then
    halt;           {Завершаем программу: файла не существует}
  close(fi);
  {Проверяем существование .BAK-файла:}
  assign(fo,name_bak);
  reset(fo);
  {$I+}
  if IOResult = 0 then
    begin
      close(fo);           {Файл .BAK существует:}
      erase(fo);           {Закрываем его}
                           {и уничтожаем}
    end;
  {Проверки закончены, подготовка к работе:}
  rename(fi,name_bak);
  reset(fi);
  assign(fo,name);
  rewrite(fo);
  .....

```

end.

Обратите внимание: проверка на существование файла **.BAK** в данном примере необходима, так как обращение

```
rename(fi,name_bak);
```

вызовет ошибку в случае, если такой файл существует.

Процедура FLUSH. Очищает внутренний буфер файла и, таким образом, гарантирует сохранность всех последних изменений файла на диске. Формат обращения:

```
FLUSH (<ф.п.>)
```

Любое обращение к файлу в Турбо Паскале осуществляется через некоторый буфер, что необходимо для согласования внутреннего представления файлового компонента (записи) с принятым в ДОС форматом хранения данных на диске. В ходе выполнения процедуры **FLUSH** все новые записи будут действительно записаны на диск. Процедура игнорируется, если файл был инициализирован для чтения процедурой **RESET**.

Функция EOF (<ф.п.>): BOOLEAN. Логическая функция, тестирующая конец файла. Возвращает **TRUE**, если файловый указатель стоит в конце файла. При записи это означает, что очередной компонент будет добавлен в конец файла, при чтении — что файл исчерпан.

Процедура CHDIR. Изменение текущего каталога. Формат обращения:

```
CHDIR (<путь>)
```

Здесь **<путь>** — строковое выражение, содержащее путь к устанавливаемому по умолчанию каталогу.

Процедура GETDIR. Позволяет определить имя текущего каталога (каталога по умолчанию). Формат обращения:

```
GETDIR (<устройство>, <каталог>)
```

Здесь **<устройство>** — выражение типа **WORD**, содержащее номер устройства: 0 — устройство по умолчанию, 1 — диск A, 2 — диск B и т.д.;

<каталог> — переменная типа **STRING**, в которую : возвращается путь к текущему каталогу на указанном диске.

Процедура MKDIR. Создает новый каталог на указанном диске. Формат обращения:

```
MKDIR (<каталог>,
```

Здесь **<каталог>** — выражение типа **STRING**, задающее путь к каталогу. Последним именем в пути, т.е. именем вновь создаваемого каталога не может быть имя уже существующего каталога.

Процедура RMDIR. Удаляет каталог. Формат обращения:

```
RMDIR (<каталог>)
```

Удаляемый каталог должен быть пустым, т.е. не содержать файлов или имен каталогов нижнего уровня.

Функция IORESULT: WORD. Возвращает условный признак последней операции ввода — вывода.

Если операция завершилась успешно, функция возвращает ноль. Коды ошибочных операций ввода — вывода представлены в прил.3. Следует помнить, что *IORESULT* становится доступной только при отключенном автоконтроле ошибок ввода — вывода. Директива компилятора *{SI-}* отключает, а директива *{SI+}* включает автоконтроль. Если автоконтроль отключен, а операция ввода — вывода привела к возникновению ошибки, устанавливается флаг ошибки и все последующие обращения к вводу — выводу блокируются, пока не будет вызвана функция *IORESULT*.

Ряд полезных файловых процедур и функций становится доступным при использовании библиотечного модуля *DOS.TPU*, входящего в стандартную библиотеку *TURBO.TPL*. Эти процедуры и функции указаны ниже. Доступ к ним возможен только после объявления *USES DOS* в начале программы (подробнее о работе с модулями см. гл.9).

Функция DISKFREE (<диск>): LONGINT. Возвращает объем в байтах свободного пространства на указанном диске. При обращении к функции выражение *<диск>* типа *BYTE* определяет номер диска: 0 — устройство по умолчанию, 1 — диск А, 2 — диск В и т.д. Функция возвращает значение -1, если указан номер несуществующего диска.

Функция DISKSIZE (<диск>): LONGINT. Возвращает полный объем диска в байтах или -1, если указан номер несуществующего диска.

Процедура FINDFIRST. Возвращает атрибуты первого из файлов, за-регистрированных в указанном каталоге. Формат обращения:

FINDFIRST (<маска>, <атрибуты>, <имя>)

Здесь *<маска>* — строковое выражение, содержащее маску файла;

<атрибуты> — выражение типа *BYTE*, содержащее уточнение к маске (атрибуты);

<имя> — переменная типа *SEARCHREC*, в которой будет возвращено имя файла.

При формировании маски файла используются следующие символы — заменители DOS:

* означает, что на месте этого символа может стоять сколько угодно (в том числе ноль) разрешенных символов имени или расширения файла;

? означает, что на месте этого символа может стоять один из разрешенных символов.

Например:

. выбирает все файлы из каталога;

c*.* выбирает все файлы с именами, начинающимися на с (*c1.pas*, *cc12345*, *c.dat* и т.д.);

a?.dat выбирает имена файлов типа *a0.dat*, *az.dat* и т.д.

Маске может предшествовать путь. Например, команда

```
c:\dir\subdir\*.pas
```

означает выбирать все файлы с расширением *.PAS* из каталога *SUBDIR*, находящегося на диске *C*; каталог *SUBDIR* зарегистрирован в каталоге верхнего уровня *DIR*, который, в свою очередь, входит в корневой каталог.

Байт *<атрибуты>* содержит двоичные разряды (биты), уточняющие, к каким именно файлам разрешен доступ при обращении к процедуре *FINDFIRST*. Вот как объявляются файловые атрибуты в модуле *DOS.TPU*:

```
const
  ReadOnly   = $01; {только чтение}
  Hidden     = $02; {скрытый файл}
  SysFile    = $04; {системный файл}
  VolumeID   = $08; {идентификатор тома}
  Directory  = $10; {имя подкаталога}
  Archive    = $20; {архивный файл}
  AnyFile    = $3F; {любой файл}
```

Комбинацией бит в этом байте можно указывать самые разные варианты, например \$06 — выбирать все скрытые и/или системные файлы.

Результат работы процедуры *FINDFIRST* возвращается в переменной типа *SEARCHREC*. Этот тип в модуле *DOS.TPU* определяется следующим образом:

```
type
  SearchRec = record
    Fill : array [1..21] of byte;
    Attr : byte;
    Time : longint;
    Size : longint;
    Name : string [12]
  end;
```

Здесь *Attr* — атрибуты файла (см. выше);

Time — время создания или последнего обновления файла; возвращается в упакованном формате; распаковать параметр можно процедурой *UNPACKTIME* (см. ниже);

Size — длина файла в байтах;

Name — имя и расширение файла.

Для распаковки параметра *TIME* используется процедура

```
UNPACKTIME(Time: LongInt; var T: DateTime).
```

В модуле *DOS.TPU* объявлен следующий тип *DateTime*:

```
type
  DateTime = record
    year : word; {год в формате 19XX}
```

```
month: word; {месяц 1..12}  
day : word; {день 1..31}  
hour : word; {час 0..23}  
min : word; {минуты 0..59}  
sec : word {секунды 0..59}  
end;
```

Результат обращения к процедуре *FINDFIRST* можно проконтролировать с помощью функции *DOSERROR* типа *WORD*, которая возвращает значения:

```
0 - нет ошибок;  
2 - не найден каталог;  
18 - каталог пуст (нет указанных файлов).
```

Процедура *FINDNEXT*. Возвращает имя следующего файла в каталоге. Формат обращения:

FINDNEXT (<сл.файл>)

Здесь <сл.файл> — запись типа *SEARCHREC* (см. выше), в которой возвращается информация о файле.

Следующая простая программа иллюстрирует способ использования процедур *FINDFIRST* и *FINDNEXT*. Программа выводит на экран список всех *PAS*-файлов текущего каталога:

```
Uses DOS;  
var  
  S: SearchRec;  
begin  
  FindFirst('*.pas',AnyFile,S);  
  while DosError = 0 do  
    begin  
      with S do  
        WriteLn(Name:12,Size:12);  
      FindNext(S)  
    end  
  end.
```

Процедура *GETFTIME*. Возвращает время создания или последнего обновления файла. Формат обращения:

GETFTIME (<ф.п.>, <время>)

Здесь <время> — переменная типа *LONGINT*, в которой возвращается время в упакованном формате.

Процедура *SETFTIME*. Устанавливает новую дату создания или обновления файла. Формат обращения:

SETFTIME (<ф.п.>, <время>)

Здесь <время> — время и дата в упакованном формате.

Упаковать запись типа *DATETIME* в переменную типа *LONGINT* можно процедурой

```
PACKTIME (var T:DateTime; var Time:LongInt).
```

(Описание типа *DATETIME* см. выше).

Процедура *GETFATTR*. Позволяет получить атрибуты файла. Формат обращения:

```
GETFATTR (<ф.п.>, <атрибуты>)
```

Здесь *<атрибуты>* — переменная типа *WORD*, в младшем байте которой возвращаются устанавливаемые атрибуты файла.

Процедура *SETFATTR*. Позволяет установить атрибуты файла. Формат обращения:

```
SETFATTR (<ф.п.>, <атрибуты>)
```

Функция *FSEARCH*: *PATHSTR*. Ищет файл в списке каталогов. Формат вызова:

```
FSEARCH (<имя>, <сп.каталогов>)
```

Здесь *<имя>* — имя отыскиваемого файла (строковое выражение или переменная типа *PATHSTR*; имени может предшествовать путь);

<сп.каталогов> — список каталогов, в которых отыскивается файл (строковое выражение или переменная типа *STRING*; имена каталогов разделяются точкой с запятой).

Результат поиска возвращается функцией *FSEARCH* в виде строки типа *PATHSTR*. В строке содержится путь и имя файла, если поиск был успешным, в противном случае возвращается пустая строка.

Тип *PATHSTR* в модуле *DOS.TPU* объявлен следующим образом:

```
type
  PathStr = String[79];
```

Следует учесть, что поиск файла всегда начинается в текущем каталоге и только после этого продолжается в тех, что перечислены в *<сп.каталогов>*. Если файл обнаружен, дальнейший поиск прекращается, даже если часть каталогов осталась непросмотренной. В частности, если файл зарегистрирован в текущем каталоге, он «заслонит» собой одноименные файлы в других каталогах.

Пусть, например, на диске имеется файл *\SUBDIR\MYFILE.PAS*. Тогда в случае, если текущий каталог — корневой, обращение

```
FSEARCH ('MYFILE.PAS', '\SUB; \SUBDIR')
```

вернет строку *\SUBDIR\MYFILE.PAS*, а обращение

```
FSEARCH ('MYFILE.PAS', '\SUB')
```

вернет пустую строку. Однако, если текущим установлен каталог *SUBDIR*, то в обоих случаях вернется строка *MYFILE.PAS* (если файл находится в текущем каталоге, в выходной строке путь к нему не указывается).

Процедура *FSPLIT*. «Расщепляет» имя файла, т.е. возвращает в качестве отдельных параметров путь к файлу, его имя и расширение. Формат обращения:

FSPLIT (<файл>, <путь>, <имя>, <расширение>)

Здесь <файл> — строковое выражение, содержащее спецификацию файла (имя с расширением и, возможно, с предшествующим путем);

<путь> — переменная типа *DIRSTR=STRING* [67], в которой возвращается путь к файлу;

<имя> — переменная типа *NAMESTR=STRING* [8], в которой возвращается имя файла;

<расширение> — переменная типа *EXTSTR=STRING* [4], в которой возвращается расширение с предшествующей ему точкой.

Процедура не проверяет наличие на диске указанного файла. В качестве входного параметра может использоваться переменная типа *PATHSTR*.

Функция *FEXPAND: PATHSTR*. Дополняет файловое имя до полной спецификации, т.е. с указанием устройства и пути. Формат вызова:

FEXPAND (<файл>)

Здесь <файл> — строковое выражение или переменная типа *PATHSTR*.

Функция не проверяет наличие указанного файла на диске, а просто дополняет имя файла недостающими параметрами — текущим устройством и путем к текущему каталогу. Результат возвращается в строке типа *PATHSTR*.

5.3. ТЕКСТОВЫЕ ФАЙЛЫ

Текстовые файлы связываются с файловыми переменными, принадлежащими стандартному типу *TEXT*. Текстовые файлы предназначены для хранения текстовой информации. Именно в такого типа файлах хранятся, например, исходные тексты программ. Компоненты (записи) текстового файла могут иметь переменную длину, что существенно влияет на характер работы с ними.

Текстовый файл трактуется в Турбо Паскале как совокупность строк переменной длины. Доступ к каждой строке возможен лишь последовательно, начиная с первой. При создании текстового файла в конце каждой записи (строки) ставится специальный признак *EOLN* (*End Of LiNe* — конец строки), а в конце всего файла — признак *EOF* (*End Of File* — конец файла). Эти признаки можно протестировать одноименными логическими функциями (см. ниже). При формировании текстовых файлов используются следующие системные соглашения:

EOLN — последовательность кодов *ASCII* #13 (CR) и #10 (LF);

EOF — код #26 стандарта *ASCII*.

Для доступа к записям применяются процедуры *READ*, *READLN*, *WRITE*, *WRITELN*. Они отличаются возможностью обращения к ним с переменным числом фактических параметров, в качестве которых могут использоваться символы, строки и числа. Первым параметром в любой из перечисленных процедур может стоять файловая переменная. В этом случае осуществляется обращение к дисковому файлу или логическому устройству, связанному с переменной процедурой *ASSIGN*. Если файловая переменная не указана, происходит обращение к стандартным файлам *INPUT* и *OUTPUT*.

Процедура *READ*. Обеспечивает ввод символов, строк и чисел. Формат обращения:

READ (<ф.п.>, <сп.ввода>) или *READ* (<сп.ввода>)

Здесь <сп.ввода> — список ввода: последовательность из одной или более переменных типа *CHAR*, *STRING*, а также любого целого или вещественного типа.

При вводе переменных типа *CHAR* выполняется чтение одного символа из файла и присваивание считанного значения переменной. Если перед выполнением чтения указатель файла достиг конца очередной строки, то результатом чтения будет символ *CR* (*ASCII* код #13), а если достигнут конец файла, то — символ *EOF* (код #26). При вводе с клавиатуры символ *CR* вводится при нажатии на клавишу *Enter*, а символ *EOF* — при одновременном нажатии клавиш *CTRL* и *Z*.

При вводе переменных типа *STRING* количество считанных процедурой и помещенных в строку символов равно максимальной длине строки, если только раньше не встретились символы *CR* или *EOF*. В этом случае сами символы *CR* и *EOF* в строку не помещаются. Если количество символов во входном потоке данных больше максимальной длины строки, «лишние» символы до конца строки отбрасываются, а новое обращение к *READ* возвращает пустую строку. Таким образом, процедура *READ* не в состоянии прочесть последовательность строк: первая строка будет прочитана нормально, а все последующие окажутся пустыми. Для ввода последовательности строк нужно использовать процедуру *READLN* (см. ниже).

При вводе числовых переменных процедура *READ* вначале выделяет подстроку во входном потоке по следующему правилу: все ведущие пробелы, символы табуляции и маркеры конца строк *EOLN* пропускаются; после выделения первого значащего символа, наоборот, любой из перечисленных символов или символ *EOF* служат признаком конца подстроки. Выделенная таким образом подстрока затем рассматривается как символическое представление числовой константы соответствующего типа и преобразуется во внутреннее представление, а полученное значение присваивается переменной. Если в подстроке был нарушен требуемый формат представления численной константы, возникает ошибка ввода-вывода. Если при пропуске ведущих пробелов встретился символ *EOF*, переменная получает значение 0. Отметим, что в Турбо Паскале не предусмотрен ввод шестнадцатеричных констант.

При использовании процедуры *READ* применительно к стандартному файлу *INPUT*, т.е. при вводе с клавиатуры, символьные строки запоминаются в буфере, который передается процедуре только после нажатия на клавишу *Enter*. Это позволяет редактировать данные при их вводе. Для редактирования используются следующие клавиши:

- *Backspace, Ctrl-H, перевод курсора влево* — стирают символ слева от курсора;
- *перевод курсора вправо* — восстанавливает символ за символом предыдущую строку ввода;
- *Ctrl-Z Enter* — завершает ввод по процедуре *READ*; оставшиеся «лишние» символьные параметры принимают значение *CHR(26)*, строки возвращаются пустыми, а численные переменные остаются без изменения.

Максимальная длина буфера ввода при работе с клавиатурой составляет 127 символов. Ввод с клавиатуры по процедуре *READ* сопровождается эхо — повтором вводимых символов на экране ПК.

Процедура *READ* прекрасно приспособлена к вводу чисел. При обращении к ней за вводом очередного целого или вещественного числа процедура «перескакивает» маркеры конца строк, т.е. фактически весь файл рассматривается ею как одна длинная строка, содержащая текстовое представление чисел. В сочетании с проверкой конца файла функцией *EOF* процедура *READ* позволяет организовать простой ввод массивов данных, например, так:

```
const
  N = 1000; {Максимальная длина ввода}
var
  f : text;
  m : array [1..N] of real;
  i : integer;
begin
  assign(f, 'prog.dat');
  reset(f);
  i := 1;
  while not EOF(f) and (i <= N) do
    begin
      read(f, m[i]);
      inc(i)
    end;
  close(f);
  .....
end.
```

Процедура READLN. Обеспечивает ввод символов, строк и чисел. Эта процедура идентична процедуре *READ* за исключением того, что после считывания последней переменной оставшаяся часть строки до маркера *EOLN* пропускается, поэтому следующее обращение к *READLN* или *READ* начинается с первого символа новой строки. Кроме того, эту процедуру

можно вызвать без параметра *<сп.вывода>* (см. процедуру *READ*), что приведет к пропуску всех символов текущей строки вплоть до *EOLN*.

Если процедура используется для чтения с клавиатуры, нажатие на клавишу *Enter* отобразится на экране как последовательность *CR + LF* и курсор будет помещен в начало следующей строки, в то время как в процедуре *READ* эхо-повтором клавиши *Enter* является символ *CR* и курсор помещается в начало текущей строки.

Процедура *WRITE*. Обеспечивает вывод информации в текстовый файл или передачу ее на логическое устройство. Формат обращения:

WRITE (*<ф.п.>*, *<сп.вывода>*) или *WRITE* (*<сп.вывода>*)

Здесь *<сп.вывода>* — список вывода: последовательность из одного или более выражений типа *CHAR*, *STRING*, *BOOLEAN*, а также любого целого или вещественного типа.

Файловая переменная *<ф.п.>*, если она указана, должна быть предварительно описана как переменная типа *TEXT* и связана с именем файла или логическим устройством процедурой *ASSIGN*. Если файловая переменная отсутствует, подразумевается вывод в стандартный файл *OUTPUT*, который обычно связан с экраном ПК.

Любой элемент списка вывода может иметь форму

OutExpr [: *MinWidth* [: *DecPlaces*]]

Здесь *OUTEXPR* — выводимое выражение;

MINWIDTH, *DECPLACES* — выражения типа *WORD* (квадратные скобки означают возможность отсутствия заключенных в них параметров).

Подпараметр *MINWIDTH*, если он присутствует, указывает минимальную ширину поля, в которое будет записываться символьное представление значения *OUTEXPR*. Если символьное представление имеет меньшую длину, чем *MINWIDTH*, оно будет дополнено слева пробелами, если — большую длину, то подпараметр *MINWIDTH* игнорируется и выводится необходимое число символов.

Подпараметр *DECPLACES* задает количество десятичных знаков в дробной части вещественного числа. Он может использоваться только совместно с *MINWIDTH* и только по отношению к выводимому выражению одного из вещественных типов.

Если ширина поля вывода не указана, соответствующий параметр выводится вслед за предыдущим без какого-либо их разделения.

Символы и строки передаются выводному файлу без изменений, но снабжаются ведущими пробелами, если задана ширина поля вывода и эта ширина больше требуемой для вывода.

При выводе логических выражений в зависимости от их значения выводятся строки *TRUE* или *FALSE*. (Ввод логических констант процедурами *READ* или *READLN* не предусмотрен).

Вещественные числа выводятся в экспоненциальном формате, если не указан подпараметр *DECPLACES*, в противном случае выбирается формат

представления числа с фиксированной точкой. Экспоненциальный формат представляет вещественное число в виде

`_s#.#####E*###,`

где:

- `_` пробел;
- `s` пробел для положительного и знак «-» для отрицательного чисел;
- `#` десятичная цифра;
- `E` символ десятичного основания;
- `*` знак «+» или «-» в зависимости от знака десятичного порядка числа.

Если подпараметр *MINWIDTH* опущен, принимается его значение по умолчанию (23). Если *MINWIDTH* меньше 10, считается, что он равен 10.

Если подпараметр *DECPLACES* равен нулю, ни дробная часть числа, ни десятичная точка не выводятся. При отрицательном значении *DECPLACES* этот параметр игнорируется и число выводится в экспоненциальном формате с учетом *MINWIDTH*. Если значение *DECPLACES* больше 18, принимается значение 18. Следует учесть, что при указании подпараметра *DECPLACES* вещественное число всегда будет выводиться в формате с фиксированной точкой и требуемым количеством знаков в дробной части, даже если значение подпараметра *MINWIDTH* окажется недостаточным для размещения целой части: в этом случае значение *MINWIDTH* автоматически увеличивается.

При выводе на экран в случае, когда длина выводимой последовательности символов превышает ширину экрана или созданного на нем окна, «лишние» символы переносятся на следующую экранную строку. При заполнении экрана или окна его содержимое сдвигается вверх на одну строку.

Процедура *WRITELN*. Эта процедура полностью идентична процедуре *WRITE* за исключением того, что выводимая строка символов завершается кодами *CR* и *LF*. При вызове *WRITELN* можно опускать параметр *<сп.вывода>*: в этом случае в файл передается маркер *EOLN*, что при выводе на экран приведет к переводу курсора в начало следующей строки.

Логическая функция *EOLN*. Возвращает *TRUE*, если во входном текстовом файле достигнут маркер конца строки. Формат обращения:

`EOLN (<ф.п.>)`

Если параметр *<ф.п.>* опущен, функция проверяет стандартный файл *INPUT*.

Существует некоторое отличие в работе функций *EOLN* и *EOF* с дисковыми файлами и логическими устройствами. Дело в том, что для логического устройства невозможно предвидеть, каким будет результат чтения очередного символа. Поэтому при работе с логическим устройством функция *EOLN* возвращает *TRUE*, если последним считанным с устройства символом был *EOLN* или *EOF*, в то время как при чтении с диска *TRUE* возвращается в случае, если следующим считываемым символом будет *EOLN* или *EOF*. Аналогичное различие наблюдается и в функции *EOF*: для

логического устройства *TRUE* возвращается в случае, если последним символом был *EOF*, а при чтении с диска — если следующим считываемым символом будет *EOF*. Иными словами, функции тестируют соответствующие признаки для логического устройства после очередного чтения, а для файла — перед чтением.

Логическая функция SEEKEOLN. Пропускает все пробелы и знаки табуляции до маркера конца строки *EOLN* или до первого значащего символа и возвращает *TRUE*, если маркер обнаружен. Формат обращения:

SEEKEOLN (<ф.п.>)

Если параметр <ф.п.> опущен, функция проверяет стандартный файл *INPUT*.

Логическая функция SEEKEOF. Пропускает все пробелы, знаки табуляции и маркеры конца строки *EOLN* до маркера конца файла или до первого значащего символа и возвращает *TRUE*, если маркер обнаружен. Формат обращения:

SEEKEOF (<ф.п.>)

Если параметр <ф.п.> опущен, функция проверяет стандартный файл *INPUT*.

В следующем примере, иллюстрирующем работу с текстовым файлом, подсчитывается общее количество символов в файле и результат делится на 40000 — таким способом можно оценить объем рукописи в так называемых учетно-издательских листах:

```
var
  f: text;
  s: String;
const
  Sum: LongInt = 0; {Здесь будет количество символов}
begin
  Write('Имя файла: ');           {Запрашиваем...}
  Readln(s);                       {и вводим имя файла}
  assign(f,s);
  Reset(f);                         {Открываем файл}
  while not EOF(f) do              {Подсчитываем...}
  begin                             {количество...}
    ReadLn(f,s);                   {символов...}
    inc(Sum, Length(s))            {в файле}
  end;
  Close(f);                         {Закрываем файл}
  WriteLn('Объем = ', Sum/40000:6:2, ' уч.изд.л.')
end.
```

5.4. ТИПИЗИРОВАННЫЕ ФАЙЛЫ

Длина любого компонента типизированного файла строго постоянна, что дает возможность организовать прямой доступ к каждому из них (т.е. доступ к компоненту по его порядковому номеру).

Перед первым обращением к процедурам ввода–вывода указатель файла стоит в его начале и указывает на первый компонент с номером 0. После каждого чтения или записи указатель сдвигается к следующему компоненту файла. Переменные в списках ввода–вывода должны иметь тот же тип, что и компоненты файла. Если этих переменных в списке несколько, указатель будет смещаться после каждой операции обмена данными между переменными и дисковым файлом.

Процедура READ. Обеспечивает чтение очередных компонентов типизированного файла. Формат обращения:

```
READ (<ф.п.>, <сп.ввода>)
```

Здесь <сп.ввода> — список ввода, содержащий одну или более переменных такого же типа, что и компоненты файла.

Файловая переменная <ф.п.> должна быть объявлена предложением **FILE OF...** и связана с именем файла процедурой **ASSIGN**. Файл необходимо открыть процедурой **RESET**. Если файл исчерпан, обращение к **READ** вызовет ошибку ввода–вывода.

Процедура WRITE. Используется для записи данных в типизированный файл. Формат обращения:

```
WRITE (<ф.п.>, <сп.вывода>)
```

Здесь <сп.вывода> — список вывода, содержащий одно или более выражений того же типа, что и компоненты файла.

Процедура SEEK. Смещает указатель файла к требуемому компоненту. Формат обращения:

```
SEEK (<ф.п.>, <N компонента>)
```

Здесь <N компонента> — выражение типа **LONGINT**, указывающее номер компонента файла.

Первый компонент файла имеет номер 0. Процедуру нельзя применять к текстовым файлам.

Функция FILESIZE. Возвращает значение типа **LONGINT**, которое содержит количество компонентов файла. Формат обращения:

```
FILESIZE (<ф.п.>)
```

Функцию нельзя использовать для текстовых файлов. Чтобы переместить указатель в конец типизированного файла, можно написать:

```
seek (FileVar, FileSize(FileVar));
```

где **FILEVAR** — файловая переменная.

Функция FILEPOS. Возвращает значение типа *LONGINT*, содержащее порядковый номер компонента файла, который будет обрабатываться следующей операцией ввода — вывода. Формат обращения:

FILEPOS (<ф.п.>)

Функцию нельзя использовать для текстовых файлов. Первый компонент файла имеет порядковый номер 0.

5.5. НЕТИПИЗИРОВАННЫЕ ФАЙЛЫ

Нетипизированные файлы объявляются как файловые переменные типа *FILE* и отличаются тем, что для них не указан тип компонентов. Отсутствие типа делает эти файлы, с одной стороны, совместимыми с любыми другими файлами, а с другой — позволяет организовать высокоскоростной обмен данными между диском и памятью.

При инициации нетипизированного файла процедурами *RESET* или *REWRITE* можно указать длину записи нетипизированного файла в байтах. Например, так:

```
var
  f: file;
begin
  .....
  assign(f, 'myfile.dat');
  reset(f, 512);
  .....
end.
```

Длина записи нетипизированного файла указывается вторым параметром при обращении к процедурам *RESET* или *REWRITE*, в качестве которого может использоваться выражение типа *WORD*. Если длина записи не указана, она принимается равной 128 байтам.

Турбо Паскаль не накладывает каких-либо ограничений на длину записи нетипизированного файла, за исключением требования полноты и ограничения максимальной длины 65535 байтами (емкость целого типа *WORD*). Однако для обеспечения максимальной скорости обмена данными следует задавать длину, которая была бы кратна длине физического сектора дискового носителя информации (512 байт). Более того, фактически пространство на диске выделяется любому файлу порциями — кластерами, которые в зависимости от типа диска могут занимать 2 и более смежных секторов. Как правило, кластер может быть прочитан или записан за один оборот диска, поэтому наивысшую скорость обмена данными можно получить, если указать длину записи, равную длине кластера.

При работе с нетипизированными файлами могут применяться все процедуры и функции, доступные типизированным файлам, за исключением *READ* и *WRITE*, которые заменяются соответственно высокоскоростными процедурами *BLOCKREAD* и *BLOCKWRITE*. Для вызова этих процедур используются следующие предложения:

BLOCKREAD (<ф.п.>,<буф>,<[,<NN>]>)

BLOCKWRITE (<ф.п.>,<буф>,<[,<NN>]>)

Здесь <буф> — буфер: имя переменной, которая будет участвовать в обмене данными с дисками;

<D> — количество записей, которые должны быть прочитаны или записаны за одно обращение к диску;

<NN> — необязательный параметр, содержащий при выходе из процедуры количество фактически обработанных записей.

За одно обращение к процедурам может быть передано до $N \cdot RECS$ байт, где $RECS$ — длина записи нетипизированного файла. Передача идет, начиная с первого байта переменной <буф>. Программист должен позаботиться о том, чтобы длина внутреннего представления переменной <буф> была достаточной для размещения всех $N \cdot RECS$ байт при чтении информации с диска. Если при чтении указана переменная <буф> недостаточной длины или если в процессе записи на диск не окажется нужного свободного пространства, возникнет ошибка ввода-вывода, которую можно заблокировать, указав необязательный параметр <NN> (переменная типа *WORD*).

После завершения процедуры указатель смещается на <NN> записей. Процедурами *SEEK*, *FILEPOS* и *FILESIZE* можно обеспечить доступ к любой записи нетипизированного файла.

Глава 6

УКАЗАТЕЛИ И ДИНАМИЧЕСКАЯ ПАМЯТЬ

6.1. ДИНАМИЧЕСКАЯ ПАМЯТЬ

Все переменные, объявленные в программе, размещаются в одной непрерывной области оперативной памяти, которая называется сегментом данных. Длина сегмента данных определяется архитектурой микропроцессоров 80x86 и составляет 65536 байт, что может вызвать известные затруднения при обработке больших массивов данных. С другой стороны, объем памяти ПК (обычно не менее 640 Кбайт) достаточен для успешного решения задач с большой размерностью данных. Выходом из положения может служить использование так называемой динамической памяти.

Динамическая память — это оперативная память ПК, предоставляемая программе при ее работе, за вычетом сегмента данных (64 Кбайт), стека (обычно 16 Кбайт) и собственно тела программы. Размер динамической памяти можно варьировать в широких пределах (см. прил.1). По умолчанию этот размер определяется всей доступной памятью ПК и, как правило, составляет не менее 200...300 Кбайт.

Динамическая память — это фактически единственная возможность обработки массивов данных большой размерности. Многие практические задачи трудно или невозможно решить без использования динамической памяти. Такая необходимость возникает, например, при разработке систем автоматизированного проектирования (САПР): размерность математических моделей, используемых в САПР, может значительно отличаться в разных проектах; статическое (т.е. на этапе разработки САПР) распределение памяти в этом случае, как правило, невозможно. Наконец, динамическая память широко используется для временного запоминания данных при работе с графическими и звуковыми средствами ПК.

Динамическое размещение данных означает использование динамической памяти непосредственно при работе программы. В отличие от этого статическое размещение осуществляется компилятором Турбо Паскаля в процессе компиляции программы. При динамическом размещении заранее не известны ни тип, ни количество размещаемых данных, к ним нельзя обращаться по именам, как к статическим переменным.

6.2. АДРЕСА И УКАЗАТЕЛИ

Оперативная память ПК представляет собой совокупность элементарных ячеек для хранения информации — байтов, каждый из которых имеет собственный номер. Эти номера называются *адресами*, они позволяют обращаться к любому байту памяти.

Турбо Паскаль предоставляет в распоряжение программиста гибкое средство управления динамической памятью — так называемые указатели.

Указатель — это переменная, которая в качестве своего значения со — держит адрес байта памяти.

В ПК адреса задаются совокупностью двух шестнадцатиразрядных слов, которые называются сегментом и смещением. *Сегмент* — это участок памяти, имеющий длину 65536 байт (64 Кбайт) и начинающийся с фи — зического адреса, кратного 16 (т.е. 0, 16, 32, 48 и т.д.). *Смещение* указывает, сколько байт от начала сегмента необходимо пропустить, чтобы обратиться к нужному адресу.

Адресное пространство ПК составляет 1 Мбайт (речь идет о так на — зываемой стандартной памяти ПК; на современных компьютерах с про — цессорами 80386 и выше адресное пространство составляет 4 Гбайт, однако в Турбо Паскале нет средств, поддерживающих работу с дополнительной памятью; при использовании среды *Borland Pascal with Objects 7.0* такая возможность имеется). Для адресации в пределах 1 Мбайта нужно 20 двоичных разрядов, которые получаются из двух шестнадцатиразрядных слов (сегмента и смещения) следующим образом (рис.6.1): содержимое сегмента смещается влево на 4 разряда, освободившиеся правые разряды заполняются нулями, результат складывается с содержимым смещения.

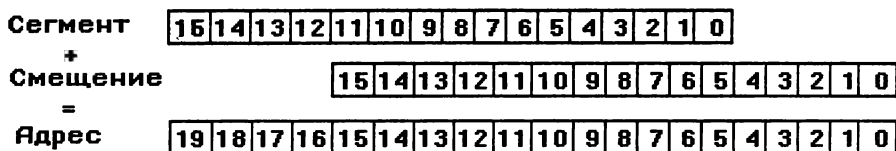


Рис.6.1. Схема формирования адреса в ПК

Фрагмент памяти в 16 байт называется *параграфом*, поэтому можно сказать, что сегмент адресует память с точностью до параграфа, а сме — щение — с точностью до байта. Каждому сегменту соответствует непре — рывная и отдельно адресуемая область памяти. Сегменты могут следовать в памяти один за другим без промежутков или с некоторым интервалом, или, наконец, перекрывать друг друга.

Таким образом, по своей внутренней структуре любой указатель представляет собой совокупность двух слов (данных типа *WORD*), трак — туемых как сегмент и смещение. С помощью указателей можно размещать в динамической памяти любой из известных в Турбо Паскале типов данных. Лишь некоторые из них (*BYTE*, *CHAR*, *SHORTINT*, *BOOLEAN*) за — нимают во внутреннем представлении один байт, остальные — несколько смежных. Поэтому на самом деле указатель адресует лишь первый байт данных.

6.3. ОБЪЯВЛЕНИЕ УКАЗАТЕЛЕЙ

Как правило, в Турбо Паскале указатель связывается с некоторым типом данных. Такие указатели будем называть *типизированными*. Для **объявления** типизированного указателя используется значок **^**, который помещается перед соответствующим типом, например:

```
var
  p1 : ^integer;
```

```

    p2 : ^real;
type
  PerconPointer = ^PerconRecord;
  PerconRecord = record
    Name : string;
    Job  : string;
    Next : PerconPointer
  end;

```

Обратите внимание: при объявлении типа *PerconPointer* мы сослались на тип *PerconRecord*, который предварительно в программе объявлен не был. Как уже отмечалось, в Турбо Паскале последовательно проводится в жизнь принцип, в соответствии с которым перед использованием какого-либо идентификатора он должен быть описан. Исключение сделано только для указателей, которые могут ссылаться на еще не объявленный тип данных. Это исключение сделано не случайно. Динамическая память дает возможность реализовать широко используемую в некоторых программах организацию данных в виде списков. Каждый элемент списка имеет в своем составе указатель на соседний элемент (рис. 6.2), что обеспечивает возможность просмотра и коррекции списка. Если бы в Турбо Паскале не было этого исключения, реализация списков была бы значительно затруднена.

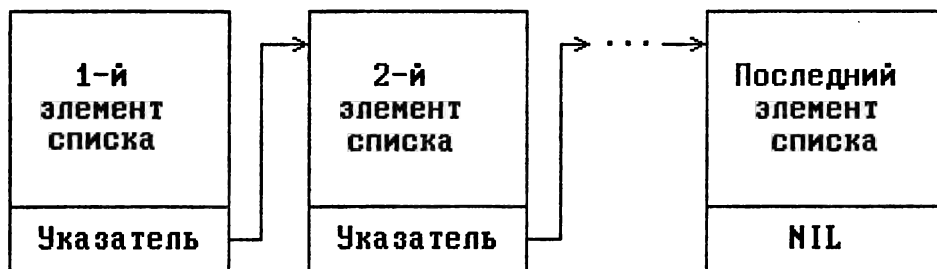


Рис.6.2. Списочная структура данных

В Турбо Паскале можно объявлять указатель и не связывать его при этом с каким-либо конкретным типом данных. Для этого служит стандартный тип *POINTER*, например:

```

var
  p: pointer;

```

Указатели такого рода будем называть *нетипизированными*. Поскольку нетипизированные указатели не связаны с конкретным типом, с их помощью удобно динамически размещать данные, структура и тип которых меняются в ходе работы программы.

Как уже говорилось, значениями указателей являются адреса переменных в памяти, поэтому следовало бы ожидать, что значение одного указателя можно передавать другому. На самом деле это не совсем так. В Турбо Паскале можно передавать значения только между указателями, связанными с одним и тем же типом данных. Если, например,

```
var
  p1, p2: ^integer;
  p3    : ^real;
  pp    : pointer;
```

то присваивание

```
p1 := p2;
```

вполне допустимо, в то время как

```
p1 := p3;
```

запрещено, поскольку *P1* и *P3* указывают на разные типы данных. Это ограничение, однако, не распространяется на нетипизированные указатели, поэтому мы могли бы записать

```
pp := p3;
p1 := pp;
```

и тем самым достичь нужного результата.

Читатель вправе задать вопрос, стоило ли вводить ограничения и тут же давать средства для их обхода. Все дело в том, что любое ограничение, с одной стороны, вводится для повышения надежности программ, а с другой — уменьшает мощность языка, делает его менее пригодным для каких-то применений. В Турбо Паскале немногочисленные исключения в отношении типов данных придают языку необходимую гибкость, но их использование требует от программиста дополнительных усилий и таким образом свидетельствует о вполне осознанном действии.

6.4. ВЫДЕЛЕНИЕ И ОСВОБОЖДЕНИЕ ДИНАМИЧЕСКОЙ ПАМЯТИ

Вся динамическая память в Турбо Паскале рассматривается как сплошной массив байтов, который называется кучей. Физически куча располагается в старших адресах сразу за областью памяти, которую занимает тело программы.

Начало кучи хранится в стандартной переменной *HEAPORG* (рис. 6.3), конец — в переменной *HEAPEND*. Текущую границу незанятой динамической памяти указывает указатель *HEAPPTR*.

Память под любую динамически размещаемую переменную выделяется процедурой *NEW*. Параметром обращения к этой процедуре является типизированный указатель. В результате обращения указатель приобретает значение, соответствующее динамическому адресу, начиная с которого можно разместить данные, например:

```
var
  i, j : ^integer;
  r    : ^real;
begin
  New(i);
  .....
end.
```

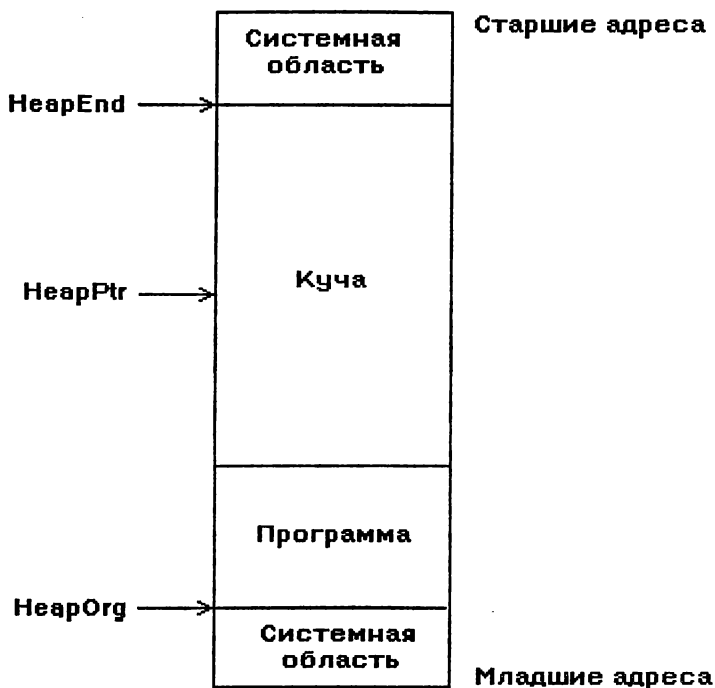


Рис.6.3. Расположение кучи в памяти ПК

После выполнения этого фрагмента указатель *I* приобретет значение, которое перед этим имел указатель кучи *HEAPPTR*, а сам *HEAPPTR* увеличит свое значение на 2, так как длина внутреннего представления типа *INTEGER*, с которым связан указатель *I*, составляет 2 байта (на самом деле это не совсем так: память под любую переменную выделяется порциями, кратными 8 байтам — см. п.6.7). Оператор

```
new(r);
```

вызовет еще раз смещение указателя *HEAPPTR*, но теперь уже на 6 байт, потому что такова длина внутреннего представления типа *REAL*. Аналогичным образом выделяется память и для переменной любого другого типа.

После того как указатель приобрел некоторое значение, т.е. стал указывать на конкретный физический байт памяти, по этому адресу можно разместить любое значение соответствующего типа. Для этого сразу за указателем без каких-либо пробелов ставится значок $\wedge$, например:

```
i^ := 2; {В область памяти i помещено значение 2}
r^ := 2*pi; {В область памяти r помещено значение 6.28}
```

Таким образом, значение, на которое указывает указатель, т.е. собственно данные, размещенные в куче, обозначаются значком $\wedge$, который ставится сразу за указателем. Если за указателем нет значка $\wedge$, то имеется в виду адрес, по которому размещены данные. Имеет смысл еще раз задуматься над только что сказанным: значением любого указателя является адрес, а чтобы указать, что речь идет не об адресе, а о тех данных, ко-

которые размещены по этому адресу, за указателем ставится $\wedge$. Если Вы четко уясните себе это, у Вас не будет проблем при работе с динамической памятью.

Динамически размещенные данные можно использовать в любом месте программы, где это допустимо для констант и переменных соответствующего типа, например:

```
r^ := sqr(r^ ) + i^ - 17;
```

Разумеется, совершенно недопустим оператор

```
r := sqr(r^ ) + i^ - 17;
```

так как указателю R нельзя присвоить значение вещественного выражения. Точно так же недопустим оператор

```
r^ := sqr(r);
```

поскольку значением указателя R является адрес, и его (в отличие от того значения, которое размещено по этому адресу) нельзя возводить в квадрат. Ошибочным будет и такое присваивание:

```
r^ := i;
```

так как вещественным данным, на которые указывает R , нельзя присвоить значение указателя (адрес).

Динамическую память можно не только забирать из кучи, но и возвращать обратно. Для этого используется процедура *DISPOSE*. Например, операторы

```
dispose(r);
dispose(i);
```

вернут в кучу 8 байт, которые ранее были выделены указателям I и R (см. выше).

Отметим, что процедура *DISPOSE(PTR)* не изменяет значения указателя PTR , а лишь возвращает в кучу память, ранее связанную с этим указателем. Однако повторное применение процедуры к свободному указателю приводит к возникновению ошибки периода исполнения. Освободившийся указатель программист может пометить зарезервированным словом *NIL*. Помечен ли какой-либо указатель или нет, можно проверить следующим образом:

```
const
  p: ^real = NIL;
begin
  .....
  if p = NIL then
    new(p);
  .....
  dispose(p);
  p := NIL;
  .....
```

end.

Никакие другие операции сравнения над указателями не разрешены.

Приведенный выше фрагмент иллюстрирует предпочтительный способ объявления указателя в виде типизированной константы (см. гл. 7) с одновременным присвоением ему значения *NIL*. Следует учесть, что начальное значение указателя (при его объявлении в разделе переменных) может быть произвольным. Использование указателей, которым не присвоено значение процедурой *NEW* или другим способом, не контролируется системой и может привести к непредсказуемым результатам.

Чередование обращений к процедурам *NEW* и *DISPOSE* обычно приводит к «ячеистой» структуре памяти. Дело в том, что все операции с кучей выполняются под управлением особой подпрограммы, которая называется *администратором кучи*. Она автоматически пристыковывается к Вашей программе компоновщиком Турбо Паскаля и ведет учет всех свободных фрагментов в куче. При очередном обращении к процедуре *NEW* эта подпрограмма отыскивает наименьший свободный фрагмент, в котором еще может разместиться требуемая переменная. Адрес начала найденного фрагмента возвращается в указателе, а сам фрагмент или его часть нужной длины помечается как занятая часть кучи. (Подробнее о работе администратора кучи см. п.6.7).

Другая возможность состоит в освобождении целого фрагмента кучи. С этой целью перед началом выделения динамической памяти текущее значение указателя *HEAPPTR* запоминается в переменной-указателе с помощью процедуры *MARK*. Теперь можно в любой момент освободить фрагмент кучи, начиная от того адреса, который запомнила процедура *MARK*, и до конца динамической памяти. Для этого используется процедура *RELEASE*. Например:

```
var
  p,p1,p2,
  p3,p4,p5 : ^integer;
begin
  new(p1);
  new(p2);
  mark(p);
  new(p3);
  new(p4);
  new(p5)
  .....
  release(p);
end.
```

В этом примере процедурой *MARK(P)* в указатель *P* было помещено текущее значение *HEAPPTR*, однако память под переменную не резервировалась. Обращение *RELEASE(P)* освободило динамическую память от помеченного места до конца кучи. Рис.6.4 иллюстрирует механизм работы процедур *NEW-DISPOSE* и *NEW-MARK-RELEASE* для рассмотренного

примера и для случая, когда вместо оператора *RELEASE(P)* используется, например, *DISPOSE(P3)*.

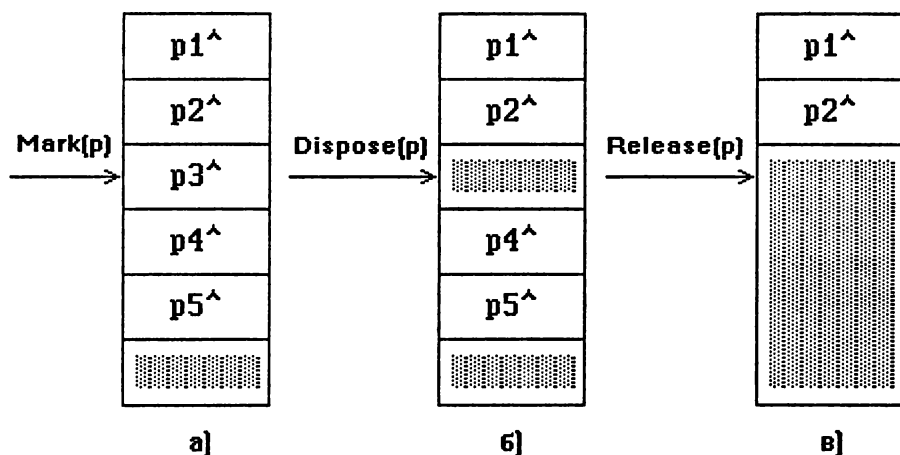


Рис.6.4. Состояние динамической памяти: а) перед освобождением; б) после *Dispose(p3)*; в) после *Release(p)*

Следует учесть, что вызов *RELEASE* уничтожает список свободных фрагментов в куче, созданных до этого процедурой *DISPOSE*, поэтому совместное использование обоих механизмов освобождения памяти в рамках одной программы не рекомендуется.

Как уже отмечалось, параметром процедуры *NEW* может быть только типизированный указатель. Для работы с нетипизированными указателями используются процедуры:

GETMEM (P, SIZE) - резервирование памяти;

FREEMEM(P, SIZE) - освобождение памяти.

Здесь *P* - нетипизированный указатель;

SIZE - размер в байтах требуемой или освобождаемой части кучи.

За одно обращение к куче процедурой *GETMEM* можно зарезервировать до 65521 байта динамической памяти.

Использование процедур *GETMEM-FREEMEM*, как и вообще вся работа с динамической памятью, требует особой осторожности и тщательного соблюдения простого правила: освобождать нужно ровно столько памяти, сколько ее было зарезервировано, и именно с того адреса, с которого она была зарезервирована.

Нетрудно обнаружить, что наличие нетипизированных указателей в Турбо Паскале (в стандартном Паскале их нет) открывает широкие возможности неявного преобразования типов. К сожалению, трудно обнаруживаемые ошибки в программе, связанные с некорректно используемыми обращениями к процедурам *NEW* и *DISPOSE*, также могут привести к нежелательному преобразованию типов. В самом деле, пусть имеется программа:

```

var
  i, j : ^integer;
  r     : ^real;
begin
  new(i);           {i := HeapOrg; HeapPtr := HeapOrg + 2}
  j := i;           {j := HeapOrg}
  j^ := 2;
  dispose(i);       {HeapPtr := HeapOrg}
  new(r);            {r := HeapOrg; HeapPtr := HeapOrg + 6}
  r^ := pi;
  WriteLn(j^);      { ?? }
end.

```

Что будет выведено на экран дисплея? Чтобы ответить на этот вопрос, проследим за значениями указателя *HEAPPTR*. Перед исполнением программы этот указатель имел значение адреса начала кучи *HEAPORG*, которое и было передано указателю *I*, а затем и *J*. После выполнения *DISPOSE(I)* указатель кучи вновь приобрел значение *HEAPORG*, этот адрес передан указателю *R* в процедуре *NEW(R)*. После того как по адресу *R* разместилось вещественное число $\pi = 3.14159$, первые 2 байта кучи оказались заняты под часть внутреннего представления этого числа. В то же время *J* все еще сохраняет адрес *HEAPORG*, поэтому оператор *WRITELN(J^)* будет рассматривать 2 байта числа π как внутреннее представление целого числа (ведь *J* — это указатель на тип *INTEGER*) и выведет 8578.

6.5. ИСПОЛЬЗОВАНИЕ УКАЗАТЕЛЕЙ

Подведем некоторые итоги. Итак, динамическая память составляет 200...300 Кбайт или больше, ее начало хранится в переменной *HEAPORG*, а конец соответствует адресу переменной *HEAPEND*. Текущий адрес свободного участка динамической памяти хранится в указателе *HEAPPTR*.

Посмотрим, как можно использовать динамическую память для размещения крупных массивов данных. Пусть, например, требуется обеспечить доступ к элементам прямоугольной матрицы 100x200 типа *EXTENDED*. Для размещения такого массива требуется память 200000 байт (100*200*10).

Казалось бы, эту проблему можно решить следующим образом:

```

var
  i, j      : integer;
  PtrArr : array [1..100, 1..200] of ^real;
begin
  for i := 1 to 100 do
    for j := 1 to 200 do
      new(PtrArr[i, j]);
    .....
  end.

```


Теперь к любому элементу вновь созданного динамического массива можно обратиться по адресу, например:

```
PtrArr[1,1]^ := 0;
  if PtrArr[i,j*2]^ > 1 then .....
```

Вспомним, однако, что длина внутреннего представления указателя составляет 4 байта, поэтому для размещения массива *PTRARR* потребуется $100 \cdot 200 \cdot 4 = 80000$ байт, что превышает размер сегмента данных (65536 байт), доступный, как уже отмечалось, программе для статического размещения данных.

Выходом из положения могла бы послужить адресная арифметика, т.е. арифметика над указателями, потому что в этом случае можно было бы отказаться от создания массива указателей *PTRARR* и вычислять адрес любого элемента прямоугольной матрицы непосредственно перед обращением к нему. Однако в Турбо Паскале над указателями не определены никакие операции, кроме операций присвоения и отношения.

Тем не менее, решить указанную задачу все-таки можно. Как мы уже знаем, любой указатель состоит из двух слов типа *WORD*, в которых хранятся сегмент и смещение. В Турбо Паскале определены две встроенные функции типа *WORD*, позволяющие получить содержимое этих слов:

SEG(X) — возвращает сегментную часть адреса;

OFS(X) — возвращает смещение.

Аргументом *X* при обращении к этим функциям может служить любая переменная, в том числе и та, на которую указывает указатель. Например, если имеем

```
var
  p : ^real;
begin
  .....
  new(p);
  p^ := 3.14;
  .....
end.
```

то функция *SEG(P)* вернет сегментную часть адреса, по которому располагается 4-байтный указатель *P*, в то время как *SEG(P^)* — сегмент 6-байтного участка кучи, в котором хранится число 3.14.

С другой стороны, с помощью встроенной функции

```
PTR(SEG,OFS: WORD): POINTER
```

можно создать значение указателя, совместимое с указателями любого типа. Таким образом возможна такая последовательность действий. Вначале процедурой *GETMEM* из кучи забираются несколько фрагментов подходящей длины (напомню, что за одно обращение к процедуре можно зарезервировать не более 65521 байт динамической памяти). Для рассматриваемого примера удобно резервировать фрагменты такой длины, чтобы в них могли, например, разместиться строки прямоугольной матрицы, т.е. $200 \cdot 10 = 2000$ байт. Начало каждого фрагмента, т.е. факти-

чески начало размещения в памяти каждой строки, запоминается в массиве *PtrStr*, состоящем из 100 указателей. Теперь для доступа к любому элементу строки нужно вычислить смещение этого элемента от начала строки и сформировать соответствующий указатель:

```

var
  i, j      : integer;
  PtrStr : array [1..100] of pointer;
  pr      : ^real;
const
  SizeOfReal = 6;
begin
  for i := 1 to 100 do
    GetMem(PtrStr[i], SizeOfReal*200);
    .....
    {Обращение к элементу матрицы [i,j]}
    pr := ptr(seg(PtrStr[i]^),
              ofs(PtrStr[i]^)+(j-1)*SizeOfReal);
    if pr^ > 1 then
      .....
  end.

```

Поскольку оператор вычисления адреса *PR := PTR...* будет, судя по всему, использоваться в программе неоднократно, полезно ввести вспомогательную функцию *GETR*, возвращающую значение элемента матрицы, и процедуру *PUTR*, устанавливающую новое значение элемента (правила объявления процедур и функций изложены в гл. 8). Каждая из них, в свою очередь, обращается к функции *ADDRR* для вычисления адреса. В примере 6.1 приводится программа, создающая в памяти матрицу из *NxM* случайных чисел и вычисляющая их среднее значение.

Пример 6.1

```

const
  SizeOfReal = 6;           {Длина переменной типа REAL}
  N = 100;                 {Количество столбцов}
  M = 200;                 {Количество строк}
var
  i, j      : integer;
  PtrStr : array [1..N] of pointer;
  s        : real;
type
  RealPoint = ^real;
  {-----}
Function AddrR(i, j: word): RealPoint;
  {По сегменту i и смещению j выдает адрес
  вещественной переменной}
begin
  AddrR := ptr(seg(PtrStr[i]^),
               ofs(PtrStr[i]^)+(j-1)*SizeOfReal)

```

```

end {AddrR};
{-----}
Function GetR(i,j: integer): real;
{Выдает значение вещественной переменной по сегменту i
и смещению j ее адреса}
begin
  GetR := AddrR(i,j)^
end {GetR};
{-----}
ProcEDURE PutR(i,j : integer; x: real);
{Помещает в переменную, адрес которой имеет сегмент i
и смещение j, вещественное значение x}
begin
  AddrR(i,j)^ := x
end {PutR};
{-----}
begin {Main}
  for i :=1 to N do
    begin
      GetMem(PtrStr[i], M*SizeOfReal);
      for j := 1 to M do PutR(i, j, Random)
    end;
    s := 0;
    for i := 1 to N do
      for j := 1 to M do
        s := s + GetR(i,j);
      WriteLn(s / (N * M) : 12:10)
    end {Main}.

```

В рассмотренном примере предполагается, что каждая строка размещается в куче, начиная с границы параграфа, и смещение для каждого указателя *PTRSTR* равно нулю. В действительности при последовательных обращениях к процедуре *GETMEM* начало очередного фрагмента следует сразу за концом предыдущего и может не попасть на границу сегмента. В результате, при размещении фрагментов максимальной длины (65521 байт) может возникнуть переполнение при вычислении смещения последнего байта.

6.6. ПРОЦЕДУРЫ И ФУНКЦИИ ДЛЯ РАБОТЫ С ДИНАМИЧЕСКОЙ ПАМЯТЬЮ

Ниже приводится описание как уже рассмотренных процедур и функций, так и некоторых других, которые могут оказаться полезными при обращении к динамической памяти.

Функция ADDR. Возвращает результат типа *POINTER*, в котором содержится адрес аргумента. Обращение:

ADDR (X)

Здесь *X* — любой объект программы (имя любой переменной, процедуры, функции). Возвращаемый адрес совместим с указателем любого типа. Отметим, что аналогичный результат возвращает операция @.

Функция CSEG. Возвращает значение, хранящееся в регистре *CS* микропроцессора (в начале работы программы в регистре *CS* содержится сегмент начала кода программы). Обращение:

CSEG

Результат возвращается в слове типа *WORD*.

Процедура DISPOSE. Возвращает в кучу фрагмент динамической памяти, который ранее был зарезервирован за типизированным указателем. Обращение:

DISPOSE (TP)

Здесь *TP* — типизированный указатель. При повторном использовании процедуры применительно к уже освобожденному фрагменту возникает ошибка периода исполнения. При освобождении динамических объектов можно указывать вторым параметром обращения к *DISPOSE* имя деструктора (подробнее см. гл.10).

Функция DSEG. Возвращает значение, хранящееся в регистре *DS* микропроцессора (в начале работы программы в регистре *DS* содержится сегмент начала данных программы). Обращение:

DSEG

Результат возвращается в слове типа *WORD*.

Процедура FREEMEM. Возвращает в кучу фрагмент динамической памяти, который ранее был зарезервирован за нетипизированным указателем. Обращение:

FREEMEM (P, SIZE)

Здесь *P* — нетипизированный указатель;

SIZE — длина в байтах освобождаемого фрагмента.

При повторном использовании процедуры применительно к уже освобожденному фрагменту возникает ошибка периода исполнения.

Процедура GETMEM. Резервирует за нетипизированным указателем фрагмент динамической памяти требуемого размера. Обращение:

GETMEM (P, SIZE)

За одно обращение к процедуре можно зарезервировать не более 65521 байтов динамической памяти. Если нет свободной памяти требуемого размера, возникает ошибка периода исполнения. Если память не фрагментирована, последовательные обращения к процедуре будут резервировать последовательные участки памяти, так что начало следующего будет располагаться сразу за концом предыдущего.

Процедура *MARK*. Запоминает текущее значение указателя кучи *HEAPPTR*. Обращение:

MARK (PTR)

Здесь *PTR* — указатель любого типа, в котором будет возвращено текущее значение *HEAPPTR*. Используется совместно с процедурой *RELEASE* для освобождения части кучи.

Функция *MAXAVAIL*. Возвращает размер в байтах наибольшего непрерывного участка кучи. Обращение:

MAXAVAIL

Результат имеет тип *LONGINT*. За один вызов процедуры *NEW* или *GETMEM* нельзя зарезервировать памяти больше, чем значение, возвращаемое этой функцией.

Функция *MEMAVAIL*. Возвращает размер в байтах общего свободного пространства кучи. Обращение:

MEMAVAIL

Результат имеет тип *LONGINT*.

Процедура *NEW*. Резервирует фрагмент кучи для размещения переменной. Обращение:

NEW (TP)

Здесь *TP* — типизированный указатель.

За одно обращение к процедуре можно зарезервировать не более 65521 байт динамической памяти. Если нет свободной памяти требуемого размера, возникает ошибка периода исполнения. Если память не фрагментирована, последовательные обращения к процедуре будут резервировать последовательные участки памяти, так что начало следующего будет располагаться сразу за концом предыдущего.

Процедура *NEW* может вызываться как функция. В этом случае параметром обращения к ней является тип переменной, размещаемой в куче, а функция *NEW* возвращает значение типа указатель. Например:

```
type
  PInt = ^integer;
var
  p: PInt;
begin
  p := New(PInt);
  ....
end.
```

При размещении в динамической памяти объекта разрешается в качестве второго параметра обращения к *NEW* указывать имя конструктора (см. гл.10).

Функция *OFS*. Возвращает значение типа *WORD*, содержащее смещение адреса указанного объекта. Вызов:

OFS (*X*)

Здесь *X* — выражение любого типа или имя процедуры.

Функция *PTR*. Возвращает значение типа *POINTER* по заданному сегменту *SEG* и смещению *OFS*. Вызов:

PTR (*SEG*, *OFS*)

Здесь *SEG* — выражение типа *WORD*, содержащее сегмент;

OFS — выражение типа *WORD*, содержащее смещение.

Значение, возвращаемое функцией, совместимо с указателем любого типа.

Процедура *RELEASE*. Освобождает участок кучи. Обращение:

RELEASE (*PTR*)

Здесь *PTR* — указатель любого типа, в котором предварительно было сохранено процедурой *MARK* значение указателя кучи. Освобождается участок кучи от адреса, хранящегося в *PTR*, до конца кучи. Одновременно уничтожается список всех свободных фрагментов, которые, возможно, были созданы процедурами *DISPOSE* или *FREEMEM*.

Функция *SEG*. Возвращает значение типа *WORD*, содержащее сегмент адреса указанного объекта. Вызов:

SEG (*X*)

Здесь *X* — выражение любого типа или имя процедуры.

Функция *SIZEOF*. Возвращает длину в байтах внутреннего представления указанного объекта. Вызов:

SIZEOF (*X*)

Здесь *X* — имя переменной, функции или типа.

Например, везде в программе из примера 6.1 вместо константы *SIZEOFREAL* можно было бы использовать обращение *SIZEOF(REAL)*.

6.7. АДМИНИСТРАТОР КУЧИ

Как уже отмечалось, администратор кучи — это служебная подпрограмма, которая обеспечивает взаимодействие пользовательской программы с кучей. Администратор кучи обрабатывает запросы процедур *NEW*, *GETMEM*, *DISPOSE*, *FREEMEM* и др. и изменяет значения указателей *HEAPPTR* и *FREELIST*. Указатель *HEAPPTR* содержит адрес нижней границы свободной части кучи, а указатель *FREELIST* — адрес описателя первого свободного блока. В модуле *SYSTEM* указатель *FREELIST* описан как *POINTER*, однако фактически он указывает на следующую структуру данных:

```

type
  PFreeRec = ^TFreeRec;
  TFreeRec = record
    Next : pointer;
    Size : pointer
  end;

```

Эта списочная структура предназначена для описания всех свободных блоков памяти, которые расположены ниже границы *HEAPPTR*. Происхождение блоков связано со случайной последовательностью использования процедур *NEW-DISPOSE* или *GETMEM-FREEMEM* («ячеистая» структура кучи). Поле *NEXT* в записи *TFREEREC* содержит адрес описателя следующего по списку свободного блока кучи или адрес, совпадающий с *HEAPEND*, если этот участок последний в списке. Поле *SIZE* содержит ненормализованную длину свободного блока или 0, если ниже адреса, содержащегося в *HEAPPTR*, нет свободных блоков. Ненормализованная длина определяется так: в старшем слове этого поля содержится количество свободных параграфов, а в младшем — количество свободных байт в диапазоне 0...15. Следующая функция преобразует значение поля *SIZE* в фактическую длину свободного блока:

```

Function BlockSize(Size: pointer): LongInt;
{Функция преобразует ненормализованную длину
свободного блока в байты}

```

```

type
  PtrRec = record
    Lo, Hi : word
  end;
var
  LengthBlock: LongInt;
begin
  BlockSize := LongInt(PtrRec(Size).Hi)*16 + PtrRec(Size).Lo
end;

```

Сразу после загрузки программы указатели *HEAPPTR* и *FREELIST* содержат один и тот же адрес, который совпадает с началом кучи (этот адрес содержится в указателе *HEAPORG*). При этом в первых 8 байтах кучи хранится запись, соответствующая типу *TFREEREC* (поле *NEXT* содержит адрес, совпадающий со значением *HEAPEND*, а поле *SIZE* — ноль, что служит дополнительным признаком отсутствия «ячеек» в динамической памяти). При работе с кучей указатели *HEAPPTR* и *FREELIST* будут иметь одинаковые значения до тех пор, пока в куче не образуется хотя бы один свободный блок ниже границы, содержащейся в указателе *HEAPPTR*. Как только это произойдет, указатель *FREELIST* станет ссылаться на начало этого блока, а в первых 8 байтах освобожденного участка памяти будет размещена запись *TFREEREC*. Используя *FREELIST* как начало списка, программа пользователя всегда сможет просмотреть весь список свободных блоков и при необходимости модифицировать его.

Описанный механизм вскрывает один не очень существенный недостаток, связанный с работой администратора кучи, а именно: в любой освободившийся блок администратор должен поместить описатель этого блока, а это означает, что длина блока не может быть меньше 8 байтов. Администратор кучи всегда выделяет память блоками, размер которых кратен размеру записи *TFREEREC*, т.е. кратен 8 байтам. Даже если программа запросит 1 байт, администратор выделит ей фактически 8 байт. Те же 8 байт будут выделены при запросе 2, 3, ..., 8 байт; при запросе 9 байт будет выделен блок в 16 байт и т.д. Это обстоятельство следует учитывать, если Вы хотите минимизировать возможные потери динамической памяти. Если запрашиваемый размер не кратен 8 байтам, в куче образуется «дырка» размером от 1 до 7 байт, причем она не может использоваться ни при каком другом запросе динамической памяти вплоть до того момента, когда связанная с ней переменная не будет удалена из кучи.

Если при очередном обращении к функции *NEW* или *GETMEM* администратор не может найти в куче нужный свободный блок, он обращается к функции, адрес которой содержит переменная *HEAPERROR*. Эта функция соответствует следующему процедурному типу:

type

```
HeapErrorFun = function (Size : word): integer;
```

Здесь *SIZE* — размер той переменной, для которой нет свободной динамической памяти. Стандартная функция, адрес которой при запуске программы содержит переменная *HEAPERROR*, возвращает 0, что приводит к останову программы по ошибке периода счета с кодом 203 (см. прил. 3). Вы можете переопределить эту функцию и таким образом блокировать останов программы. Для этого необходимо написать собственную функцию и поместить ее адрес в указатель *HEAPERROR*. Например:

```
Function HeapFunc(Size: Word): integer; far;
begin
  HeapFunc := 1
end;
begin {Основная программа}
  HeapError := @HeapFunc;
  .....
end.
```

Отметим, что функция типа *HEAPERRORFUN* вызывается только в том случае, когда обращение с требованием выделения динамической памяти было неуспешным. Она может возвращать одно из трех значений:

- 0 — прекратить работу программы;
- 1 — присвоить соответствующему указателю значение *NIL* и продолжить работу программы;
- 2 — повторить выделение памяти; разумеется, в этом случае внутри функции типа *HEAPERRORFUN* необходимо освободить память нужного размера.

Глава 7

ТИПИЗИРОВАННЫЕ КОНСТАНТЫ

В Турбо Паскале допускается использование типизированных констант. Они задаются в разделе объявления констант следующим образом:

<идентификатор> : <тип> = <значение>

Здесь <идентификатор> — идентификатор константы;

<тип> — тип константы;

<значение> — значение константы.

Типизированным константам можно присваивать другие значения в ходе выполнения программы, поэтому фактически они представляют собой переменные с начальными значениями. Типизированная константа приобретает указанное в ее объявлении значение, т.е. иницируется, лишь один раз: к моменту начала работы программы. При повторном входе в блок (процедуру или функцию), в котором она объявлена, инициация типизированной константы не производится и она сохраняет то значение, которое имела к моменту выхода из блока.

Типизированные константы могут быть любого типа, кроме файлов. Нельзя также объявить типизированную константу — запись, если хотя бы одно из ее полей является полем файлового типа.

Поскольку типизированная константа фактически не отличается от переменной, ее нельзя использовать в качестве значения при объявлении других констант или границ типа — диапазона.

7.1. КОНСТАНТЫ ПРОСТЫХ ТИПОВ И ТИПА *STRING*

Объявление таких констант обычно не вызывает трудностей, так как в качестве их значения используются нетипизированные константы или их идентификаторы.

Примеры объявлений:

type

colors = (white, red, black);

{ ----- Правильные объявления: ----- }

const

CurrCol : colors = red;

name : **string** = 'Вирт Н.';

year : word = 1989;

x : real = 0.1;

min : integer = 0;

max : integer = 10;

days : 1..31 = 1;

answer : char = 'Y';

{ ----- Неправильные объявления: ----- }

mass : **array** [min..max] **of** real; {Нельзя использовать

типизированные константы в качестве границ диапазона;

a,b,c : byte = 0; {Нельзя использовать


```

      ((12,13), (14,15)))));
begin
  for i := 1 to 3 do
    for j := 1 to 5 do Write(matr[i,j]:3);
  writeln;
  for i := 0 to 1 do
    for j := 0 to 1 do
      for k := 0 to 2 do Write(cube[i,j,k]:3);
    writeln;
  for i := 0 to 1 do
    for j := 0 to 1 do
      for k := 0 to 1 do
        for l := 0 to 1 do Write(mas4[i,j,k,l]:3);
      writeln;
    end.
end.

```

Количество переменных в списке констант должно строго соответствовать объявленной длине массива по каждому измерению.

7.3. КОНСТАНТЫ-ЗАПИСИ

Определение константы — записи имеет следующий вид:

<идентификатор> : <тип> = (<сп.знач.полей>)

Здесь <идентификатор> — идентификатор константы;

<тип> — тип записи;

<сп.знач.полей> — список значений полей.

Список значений полей представляет собой список из последовательностей вида: имя поля, двоеточие и константа. Элементы списка отделяются друг от друга двоеточиями, например:

```

type
  point = record
    x, y : real
  end;
  vect = array [0..1] of point;
  month = (Jan, Feb, Mar, Apr, May, Jun,
    Jly, Aug, Sep, Oct, Nov, Dec);
  date = record
    d : 1..31;
    m : month;
    y : 1900..1999
  end;

const
  origon : point = (x : 0; y : -1);
  line : vector =
    ((x : -3.1; y : 1.5),
    (x : 5.9; y : 3.0));
  SomeDay : date = (d : 16; m : Mar; y : 1989);

```

Поля должны указываться в той последовательности, в какой они перечислены в объявлении типа. Если в записи используется хотя бы одно поле файлового типа, такую запись нельзя объявить типизированной константой. Для записей с вариантными полями указывается только один из возможных вариантов констант. Например:

```

type
  forma = record
    case Boolean of
      true  : (BirthPlace : string [40]);
      false : (Country    : string [20];
               EntryPort  : string [20];
               EntryDate   : array [1..3] of word;
               count       : word)
    end;
const
  Percon1 : forma = (Country    : 'Норвегия';
                     EntryPort  : 'Мурманск';
                     EntryDate   : (16, 3, 89);
                     count       : 12);
  Percon2 : forma = (BirthPlace : 'Москва');
```

7.4. КОНСТАНТЫ-МНОЖЕСТВА

Значение типизированной константы-множества задается в виде правильного конструктора множества, например:

```

type
  days    = set of 1..31;
  digc    = set of '0'..'9';
  error   = set of 1..24;
const
  WorkDays : days = [1..5, 8..12, 15..19, 22..26, 29, 30];
  EvenDigits: digc = ['0', '2', '4', '6', '8'];
  ErrorFlag : error = [];
```

7.5. КОНСТАНТЫ-УКАЗАТЕЛИ

Единственным значением типизированной константы-указателя может быть только *NIL*, например:

```

const
  pr : ^real = NIL;
```

Глава 8

ПРОЦЕДУРЫ И ФУНКЦИИ

Процедуры и функции, как отмечалось в гл.2, представляют собой от — носительно самостоятельные фрагменты программы, оформленные особым образом и снабженные именем. Упоминание этого имени в тексте про — граммы называется *вызовом процедуры* (функции). Отличие функции от процедуры заключается в том, что результатом исполнения операторов, образующих тело функции, всегда является некоторое единственное значение или указатель, поэтому обращение к функции можно исполь — зовать в соответствующих выражениях наряду с переменными и кон — стантами. Условимся далее называть процедуру или функцию общим именем «подпрограмма», если только для излагаемого материала указанное отличие не имеет значения.

Подпрограммы представляют собой инструмент, с помощью которого любая программа может быть разбита на ряд в известной степени неза — висимых друг от друга частей. Такое разбиение необходимо по двум причинам.

Во — первых, это средство экономии памяти: каждая подпрограмма существует в программе в единственном экземпляре, в то время как об — ращаться к ней можно многократно из разных точек программы. При вызове подпрограммы активизируется последовательность образующих ее операторов, а с помощью передаваемых подпрограмме параметров нужным образом модифицируется реализуемый в ней алгоритм.

Вторая причина заключается в применении методики нисходящего проектирования программ (см. гл.2). В этом случае алгоритм представ — ляется в виде последовательности относительно крупных подпрограмм, реализующих более или менее самостоятельные смысловые части алго — ритма. Подпрограммы в свою очередь могут разбиваться на менее крупные подпрограммы нижнего уровня и т.д. (рис. 8.1). Последовательное струк — турирование программы продолжается до тех пор, пока реализуемые подпрограммами алгоритмы не станут настолько простыми, чтобы их можно было легко запрограммировать.

В этой главе подробно рассматриваются все аспекты использования подпрограмм в Турбо Паскале.

8.1. ЛОКАЛИЗАЦИЯ ИМЕН

Напомню, что вызов подпрограммы осуществляется простым упоми — нанием имени процедуры в операторе вызова процедуры или имени функции в выражении. При использовании расширенного синтаксиса Турбо Паскаля (см. ниже) функции можно вызывать точно так же, как и процедуры. Как известно, любое имя в программе должно быть обяза — тельно описано перед тем как оно появится среди исполняемых опера — торов. Не делается исключения и в отношении подпрограмм: каждую свою

процедуру и функцию программисту необходимо описать в разделе описаний.

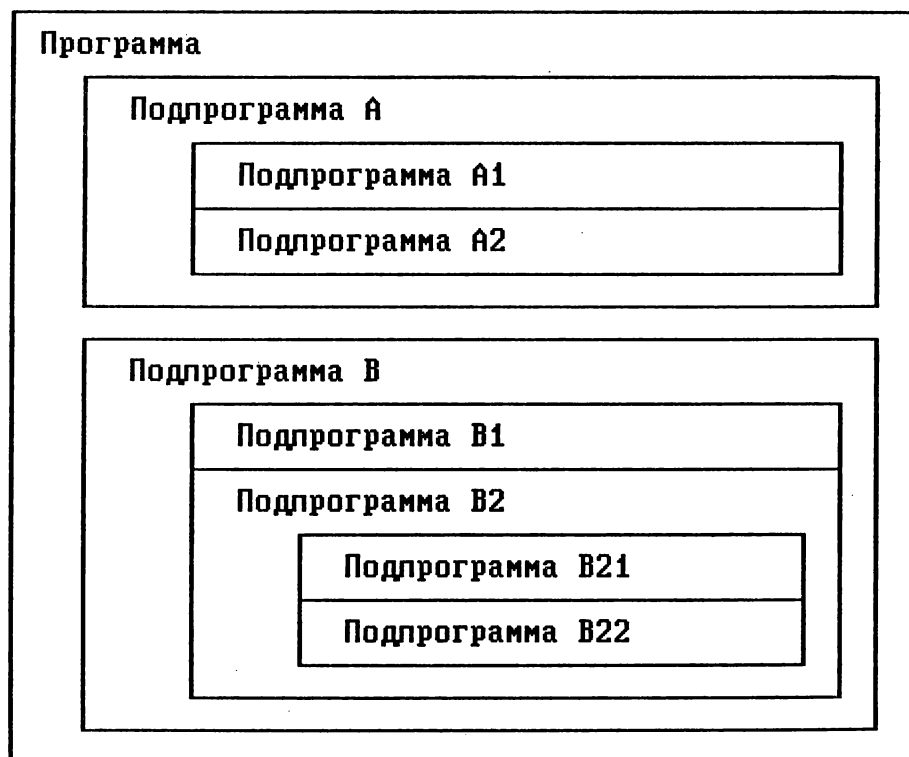


Рис.8.1. Пример структуры программы

Описать подпрограмму — это значит указать ее заголовок и тело. В заголовке объявляются имя подпрограммы и формальные параметры, если они есть. Для функции, кроме того, указывается тип возвращаемого ею результата. За заголовком следует тело подпрограммы, которое, подобно программе, состоит из раздела описаний и раздела исполняемых операторов. В разделе описаний подпрограммы могут встретиться описания подпрограмм низшего уровня, в тех — описания других подпрограмм и т.д.

Вот какую иерархию описаний получим, например, для программы, структура которой изображена на рис.8.1 (для простоты считается, что все подпрограммы представляют собой процедуры без параметров):

```

Program ...;
Procedure A;
  Procedure A1;
  .....
  begin
    ....
    end {A1};
  Procedure A2;

```

```

.....
begin
  ....
  end {A2};
begin {A}
  .....
  end {A};
Procedure B;
Procedure B1;
.....
begin
  ....
  end {B1};
Procedure B2;
  Procedure B21;
.....

```

и т.д.

Подпрограмма любого уровня имеет обычно множество имен констант, переменных, типов и вложенных в нее подпрограмм низшего уровня. Считается, что все имена, описанные внутри подпрограммы, локализируются в ней, т.е. они как бы «невидимы» снаружи подпрограммы. Таким образом, со стороны операторов, использующих обращение к подпрограмме, она трактуется как «черный ящик», в котором реализуется тот или иной алгоритм. Все детали этой реализации скрыты от глаз пользователя подпрограммы и потому недоступны ему. Например, в рассмотренном выше примере из основной программы можно обратиться к процедурам *A* и *B*, но нельзя вызвать ни одну из вложенных в них процедур *A1*, *A2*, *B1* и т.д.

Сказанное относится не только к именам подпрограмм, но и вообще к любым именам, объявленным в них — типам, константам, переменным и меткам. Все имена в пределах подпрограммы, в которой они объявлены, должны быть уникальными и не могут совпадать с именем самой подпрограммы.

При входе в подпрограмму низшего уровня становятся доступными не только объявленные в ней имена, но и сохраняется доступ ко всем именам верхнего уровня. Образно говоря, любая подпрограмма как бы окружена полупрозрачными стенками: снаружи подпрограммы мы не видим ее внутренности, но, попав в подпрограмму, можем наблюдать все, что делается снаружи. Так, например, из подпрограммы *B21* мы можем вызвать подпрограмму *A*, использовать имена, объявленные в основной программе, в подпрограммах *B* и *B2*, и даже обратиться к ним. Любая подпрограмма может, наконец, вызвать саму себя — такой способ вызова называется рекурсией.

Пусть имеем такое описание:

```

Program ...;
var V1 : ...;
  Procedure A;
    var V2 : ...;

```

```
.....  
end {A};  
Procedure B;  
var V3 : ...;  
  Procedure B1;  
    var V4 : ...;  
      Procedure B11;  
        var V5;  
          .....  
        .....
```

Из процедуры *B11* доступны все пять переменных *V1,...,V5*, из процедуры *B1* доступны переменные *V1,...,V4*, из центральной программы — только *V1*.

При взаимодействии подпрограмм одного уровня иерархии вступает в силу основное правило Турбо Паскаля: любая подпрограмма перед ее использованием должна быть описана. Поэтому из подпрограммы *B* можно вызвать подпрограмму *A*, но из *A* вызвать *B* невозможно (точнее, такая возможность появляется только с использованием опережающего описания, см. п.8.6.) Продолжая образное сравнение, подпрограмму можно уподобить ящику с непрозрачными стенками и дном и полупрозрачной крышкой: из подпрограммы можно смотреть только «вверх» и нельзя «вниз», т.е. подпрограмме доступны только те объекты верхнего уровня, которые описаны до описания данной подпрограммы. Эти объекты называются *глобальными* по отношению к подпрограмме.

В отличие от стандартного Паскаля в Турбо Паскале допускается произвольная последовательность описания констант, переменных, типов, меток и подпрограмм. Например, раздел **VAR** описания переменных может появляться в пределах раздела описаний одной и той же подпрограммы много раз и перемежаться с объявлениями других объектов и подпрограмм. Для Турбо Паскаля совершенно безразличен порядок следования и количество разделов **VAR**, **CONST**, **TYPE**, **LABEL**, но при определении области действия этих описаний следует помнить, что имена, описанные ниже по тексту программы, недоступны из ранее описанных подпрограмм, например:

```
var V1 : ...;  
  Procedure S;  
    var V2 : ...;  
    .....  
  end {S};  
var V3 : ...,  
  .....
```

Из процедуры *S* можно обратиться к переменным *V1* и *V2*, но нельзя использовать *V3*, так как описание *V3* следует в программе за описанием процедуры *S*.

Имена, локализованные в подпрограмме, могут совпадать с ранее объявленными глобальными именами. В этом случае считается, что локальное имя «закрывает» глобальное и делает его недоступным, например:


```

var
  i : integer;
Procedure P;
var
  i : integer;
begin
  writeln(i)
end {P};
begin
  i := 1;
  P
end.

```

Что напечатает эта программа? Все, что угодно: значение внутренней переменной *I* при входе в процедуру *P* не определено, хотя одноименная глобальная переменная имеет значение 1. Локальная переменная «закроет» глобальную и на экран будет выведено произвольное значение, содержащееся в неинициализированной внутренней переменной. Если убрать описание

```

var
  i : integer;

```

из процедуры *P*, то на экран будет выведено значение глобальной переменной *I*, т.е. 1.

Таким образом, одноименные глобальные и локальные переменные — это разные переменные. Любое обращение к таким переменным в теле подпрограммы трактуется как обращение к локальным переменным, т.е. глобальные переменные в этом случае попросту недоступны.

8.2. ОПИСАНИЕ ПОДПРОГРАММЫ

Описание подпрограммы состоит из заголовка и тела подпрограммы.

8.2.1. Заголовок

Заголовок процедуры имеет вид:

```
PROCEDURE <имя> [( <сп.ф.п.> )];
```

Заголовок функции:

```
FUNCTION <имя> [( <сп.ф.п.> )] : <тип>;
```

Здесь <имя> — имя подпрограммы (правильный идентификатор);

<сп.ф.п.> — список формальных параметров;

<тип> — тип возвращаемого функцией результата.

Сразу за заголовком подпрограммы может следовать одна из стандартных директив **ASSEMBLER**, **EXTERNAL**, **FAR**, **FORWARD**, **INLINE**, **INTERRUPT**, **NEAR**. Эти директивы уточняют действия компилятора и распространяются на всю подпрограмму и только на нее, т.е. если за подпрограммой следует другая подпрограмма, стандартная директива, указанная за заголовком первой, не распространяется на вторую.

ASSEMBLER — эта директива отменяет стандартную последовательность машинных инструкций, вырабатываемых при входе в процедуру и перед выходом из нее. Тело подпрограммы в этом случае должно реализовываться с помощью команд встроенного ассемблера (см. п.11.8).

EXTERNAL — с помощью этой директивы объявляется внешняя подпрограмма (см. п.11.1).

FAR — компилятор должен создавать код подпрограммы, рассчитанный на дальнюю модель вызова. Директива **NEAR** заставит компилятор создать код, рассчитанный на ближнюю модель памяти. По умолчанию все подпрограммы, объявленные в интерфейсной части модулей, генерируются с расчетом на дальнюю модель вызова, а все остальные подпрограммы — на ближнюю модель.

В соответствии с архитектурой микропроцессора ПК, в программах могут использоваться две модели памяти: ближняя и дальняя. Модель памяти определяет возможность вызова процедуры из различных частей программы: если используется *ближняя модель*, вызов возможен только в пределах 64 Кбайт (в пределах одного сегмента кода, который выделяется основной программе и каждому используемому в ней модулю); при *дальней модели* вызов возможен из любого сегмента. Ближняя модель экономит один байт и несколько микросекунд на каждом вызове подпрограммы, поэтому стандартный режим компиляции предполагает эту модель памяти. Однако при передаче процедурных параметров (см. п.8.4), а также в оверлейных модулях (см. п.11.6) соответствующие подпрограммы должны компилироваться с расчетом на универсальную — дальнюю — модель памяти, одинаково пригодную при любом расположении процедуры и вызывающей ее программы в памяти.

Явное объявление модели памяти стандартными директивами имеет более высокий приоритет по сравнению с опциями настройки среды Турбо Паскаля.

FORWARD — используется при опережающем описании (см. п.8.6) для сообщения компилятору, что описание подпрограммы следует где-то дальше по тексту программы (но в пределах текущего программного модуля).

INLINE — указывает на то, что тело подпрограммы реализуется с помощью встроенных машинных инструкций (см. п.11.2).

INTERRUPT — используется при создании процедур обработки прерываний (см. п.11.4).

8.2.2. Параметры

Список формальных параметров необязателен и может отсутствовать. Если же он есть, то в нем должны быть перечислены имена формальных параметров и их типы, например:

Procedure SB(a: real; b: integer; c: char ;

Как видно из примера, параметры в списке отделяются друг от друга точками с запятой. Несколько следующих подряд однотипных параметров можно объединять в подсписки, например, вместо

```
Function F(a: real; b: real): real;
```

можно написать проще:

```
Function F(a,b: real): real;
```

Операторы тела подпрограммы рассматривают список формальных параметров как своеобразное расширение раздела описаний: все переменные из этого списка могут использоваться в любых выражениях внутри подпрограммы. Таким способом осуществляется настройка алгоритма подпрограммы на конкретную задачу.

Рассмотрим следующий пример. В языке Турбо Паскаль нет операции возведения в степень, однако с помощью встроенных функций *LN(X)* и *EXP(X)* нетрудно реализовать новую функцию с именем, например, *POWER*, осуществляющую возведение любого вещественного числа в любую вещественную степень. В программе (пример 8.1) вводится пара чисел *X* и *Y* и выводится на экран дисплея результат возведения *X* сначала в степень $+Y$, а затем $-Y$. Для выхода из программы нужно ввести *Ctrl-Z* и *Enter*.

Пример 8.1

```
var
  x,y : real;
  {-----}
Function Power(a,b : real): real;
begin {Power}
  if a > 0 then
    Power := exp(b * ln(a))
  else if a < 0 then
    Power := exp(b * ln(abs(a)))
  else if b = 0 then
    Power := 1
  else
    Power := 0
end {Power};
  {-----}
begin {main}
  repeat
    readln(x,y);
    writeln(Power(x,y):12:10, Power(x,-y):15:10)
  until EOF
end {main}.
```

Для вызова функции *POWER* мы просто указали ее в качестве параметра при обращении к встроенной процедуре *WRITELN*. Параметры *X* и *Y* в момент обращения к функции — это фактические параметры. Они

подставляются вместо формальных параметров *A* и *B* в заголовке функции и затем над ними осуществляются нужные действия. Полученный результат присваивается идентификатору функции — именно он и будет возвращен как значение функции при выходе из нее. В программе функция *POWER* вызывается дважды — сначала с параметрами *X* и *Y*, а затем *X* и $-Y$, поэтому будут получены два разных результата.

Механизм замены формальных параметров на фактические позволяет нужным образом настроить алгоритм, реализованный в подпрограмме. Турбо Паскаль следит за тем, чтобы количество и тип формальных параметров строго соответствовали количеству и типам фактических параметров в момент обращения к подпрограмме. Смысл используемых фактических параметров зависит от того, в каком порядке они перечислены при вызове подпрограммы. В примере 8.1 первый по порядку фактический параметр будет возводиться в степень, задаваемую вторым параметром, а не наоборот. Пользователь должен сам следить за правильным порядком перечисления фактических параметров при обращении к подпрограмме.

Любой из формальных параметров подпрограммы может быть либо параметром — значением, либо параметром — переменной, либо, наконец, параметром — константой.

В предыдущем примере параметры *A* и *B* определены как параметры — значения. Если параметры определяются как параметры — переменные, перед ними необходимо ставить зарезервированное слово *VAR*, а если это параметры — константы, — слово *CONST*, например:

```
Procedure MyProcedure(var a: real; b: real; const c: String);
```

Здесь *A* — параметр — переменная, *B* — параметр — значение, а *C* — параметр — константа.

Определение формального параметра тем или иным способом существенно, в основном, только для вызывающей программы: если формальный параметр объявлен как параметр — переменная, то при вызове подпрограммы ему должен соответствовать фактический параметр в виде переменной нужного типа; если формальный параметр объявлен как параметр — значение или параметр — константа, то при вызове ему может соответствовать произвольное выражение. Контроль за неукоснительным соблюдением этого правила осуществляется компилятором Турбо Паскаля. Если бы для предыдущего примера был использован такой заголовок функции:

```
Function Power (var a, b : real) : real;
```

то при втором обращении к функции компилятор указал бы на несоответствие типа фактических и формальных параметров (параметр $-Y$ есть выражение, в то время как соответствующий ему формальный параметр описан как параметр — переменная).

Для того чтобы понять, в каких случаях использовать тот или иной тип параметров, рассмотрим, как осуществляется замена формальных параметров на фактические в момент обращения к подпрограмме.

Если параметр определен как параметр—значение, то перед вызовом подпрограммы это значение вычисляется, полученный результат копируется во временную память и передается подпрограмме. Важно учесть, что даже если в качестве фактического параметра указано простейшее выражение в виде переменной или константы, все равно подпрограмме будет передана лишь копия переменной (константы). Любые возможные изменения в подпрограмме параметра—значения никак не воспринимаются вызывающей программой, так как в этом случае изменяется копия фактического параметра.

Если параметр определен как параметр—переменная, то при вызове подпрограммы передается сама переменная, а не ее копия (фактически в этом случае подпрограмме передается *адрес* переменной). Изменение параметра—переменной приводит к изменению самого фактического параметра в вызывающей программе.

В случае параметра—константы в подпрограмму также передается адрес области памяти, в которой располагается переменная или вычисленное значение. Однако компилятор блокирует любые присваивания параметру—константе нового значения в теле подпрограммы.

Представленный ниже пример 8.2 поясняет изложенное. В программе задаются два целых числа 5 и 7, эти числа передаются процедуре *INC2*, в которой они удваиваются. Один из параметров передается как параметр—переменная, другой — как параметр—значение. Значения параметров до и после вызова процедуры, а также результат их удвоения выводятся на экран.

Пример 8.2

```

const
  a : integer = 5;
  b : integer = 7;
{-----}
Procedure Inc2 (var c: integer; b: integer);
begin {Inc2}
  c := c + c;
  b := b + b;
  WriteLn('удвоенные :', c:5, b:5)
end {inc2};
{-----}
begin {main}
  WriteLn('исходные  :', a:5, b:5);
  Inc2(a,b);
  WriteLn('результат :', a:5, b:5)
end {main}.

```

В результате прогона программы будет выведено:

исходные	:	5	7
удвоенные	:	10	14
результат	:	10	7

Как видно из примера, удвоение второго формального параметра в процедуре *INC2* не вызвало изменения фактической переменной *B*, так как этот параметр описан в заголовке процедуры как параметр — значение. Этот пример может служить еще и иллюстрацией механизма «закрывания» глобальной переменной одноименной локальной: хотя переменная *B* объявлена как глобальная (она описана в вызывающей программе перед описанием процедуры), в теле процедуры ее «закрывает» локальная переменная *B*, объявленная как параметр — значение.

Итак, параметры — переменные используются как средство связи алгоритма, реализованного в подпрограмме, с внешним миром: с помощью этих параметров подпрограмма может передавать результаты своей работы вызывающей программе. Разумеется, в распоряжении программиста всегда есть и другой способ передачи результатов — через глобальные переменные. Однако злоупотребление глобальными связями делает программу, как правило, запутанной, трудной в понимании и сложной в отладке. В соответствии с требованиями хорошего стиля программирования рекомендуется там, где это возможно, использовать передачу результатов через фактические параметры — переменные.

С другой стороны, описание всех формальных параметров как параметров — переменных нежелательно по двум причинам. Во — первых, это исключает возможность вызова подпрограммы с фактическими параметрами в виде выражений, что делает программу менее компактной. Во — вторых, и главных, в подпрограмме возможно случайное использование формального параметра, например, для временного хранения промежуточного результата, т.е. всегда существует опасность непреднамеренно испортить фактическую переменную. Вот почему параметрами — переменными следует объявлять только те, через которые подпрограмма в действительности передает результаты вызывающей программе. Чем меньше параметров объявлено параметрами — переменными и чем меньше в подпрограмме используется глобальных переменных, тем меньше опасность получения непредусмотренных программистом побочных эффектов, связанных с вызовом подпрограммы, тем проще программа в понимании и отладке. По той же причине не рекомендуется использовать параметры — переменные в заголовке функции: если результатом работы функции не может быть единственное значение, то логичнее использовать процедуру или нужным образом декомпозировать алгоритм на несколько подпрограмм.

Существует еще одно обстоятельство, которое следует учитывать при выборе вида формальных параметров. Как уже говорилось, при объявлении параметра — значения осуществляется копирование фактического параметра во временную память. Если этим параметром будет массив большой размерности, то существенные затраты времени и памяти на копирование при многократных обращениях к подпрограмме можно минимизировать, объявив этот параметр параметром — константой. Параметр — константа не копируется во временную область памяти, что сокращает затраты времени на вызов подпрограммы, однако любые его изменения в теле подпрограммы невозможны — за этим строго следит компилятор.

8.3. ПАРАМЕТРЫ-МАССИВЫ И ПАРАМЕТРЫ-СТРОКИ

Может сложиться впечатление, что объявление переменных в списке формальных параметров подпрограммы ничем не отличается от объявления их в разделе описания переменных. Действительно, в обоих случаях много общего, но есть одно существенное различие: типом любого параметра в списке формальных параметров может быть только стандартный или ранее объявленный тип. Поэтому нельзя, например, объявить следующую процедуру:

```
Procedure S (a: array [1..10] of real);
```

так как в списке формальных параметров фактически объявляется тип-диапазон, указывающий границы индексов массива.

Если мы хотим передать какой-то элемент массива, то проблем, как правило, не возникает, но если в подпрограмму передается весь массив, то следует первоначально описать его тип. Например:

```
type
  atype = array [1..10] of real;
Procedure S (a: atype);
  .....
```

Поскольку строка является фактически своеобразным массивом, ее передача в подпрограмму осуществляется аналогичным образом:

```
type
  intype = string [15];
  outype = string [30];
Function St (s : intype) : outype;
  .....
```

Требование описать любой тип-массив или тип-строку перед объявлением подпрограммы на первый взгляд кажется несущественным. Действительно, в рамках простейших вычислительных задач обычно заранее известна структура всех используемых в программе данных, поэтому статическое описание массивов не вызывает проблем. Однако разработка программных средств универсального назначения связана со значительными трудностями.

По существу, речь идет о том, что в Турбо Паскале невозможно использовать в подпрограммах массивы с «плавающими» границами изменения индексов. Например, если разработана программа, обрабатывающая матрицу 10 x 10 элементов, то для обработки матрицы 9 x 11 элементов необходимо переопределить тип, т.е. перекомпилировать всю программу (речь идет не о динамическом размещении массивов в куче, а о статическом описании массивов и передаче их как параметров в подпрограммы). Этот недостаток, как и отсутствие в языке средств обработки исключительных ситуаций (прерываний), унаследован из стандартного Паскаля и представляет собой объект постоянной и вполне заслуженной его критики. Разработчики Турбо Паскаля не рискнули кардинально изменить свойства

базового языка, но, тем не менее, включили в него некоторые средства, позволяющие в известной степени смягчить отмеченные недостатки¹.

Прежде всего, в среде Турбо Паскаля можно установить режим компиляции, при котором отключается контроль за совпадением длины фактического и формального параметра — строки (см. прил.1). Это позволяет легко решить вопрос о передаче подпрограмме строки произвольной длины. При передаче строки меньшего размера формальный параметр будет иметь ту же длину, что и параметр обращения; передача строки большего размера приведет к ее усечению до максимального размера формального параметра. Следует сказать, что контроль включается только при передаче строки, объявленной как формальный параметр — переменная. Если соответствующий параметр объявлен параметром — значением, эта опция игнорируется и длина не контролируется.

Значительно сложнее обстоит дело с передачей массивов произвольной длины. Наиболее универсальным приемом в этом случае будет, судя по всему, работа с указателями и использование индексной арифметики. Несколько проще можно решить эту проблему при помощи нетипизированных параметров (см. п.8.5). В версии Турбо Паскаля 7.0 язык поддерживает так называемые открытые массивы, легко решающие проблему передачи подпрограмме одномерных массивов переменной длины.

Открытый массив представляет собой формальный параметр подпрограммы, описывающий базовый тип элементов массива, но не определяющий его размерности и границы:

Procedure MyProc (OpenArray: **array of Integer**);

Внутри подпрограммы такой параметр трактуется как одномерный массив с нулевой нижней границей. Верхняя граница открытого массива возвращается функцией *HIGH*, упоминавшейся в п.4.1.1. Используя 0 как минимальный индекс и значение, возвращаемое функцией *HIGH*, как максимальный индекс, подпрограмма может обрабатывать одномерные массивы произвольной длины:

{Иллюстрация использования открытых массивов:
программа выводит на экран содержимое двух одномерных массивов разной длины с помощью одной процедуры ArrayPrint}

Procedure ArrayPrint(aArray: **array of Integer**);

var

k: Integer;

begin

for k := 0 to High(aArray) do

Write(aArray[k]:8);

WriteLn

end;

const

A: **array** [-1..2] **of Integer** = (0,1,2,3);

B: **array** [5..7] **of Integer** = (4,5,6);

¹ Эти недостатки практически полностью устранены в языке *Object Pascal*, используемом в визуальной среде программирования *Delphi*.


```
begin
    ArrayPrint(A);
    ArrayPrint(B)
end.
```

Как видно из этого примера, фактические границы массивов *A* и *B*, передаваемых в качестве параметров вызова процедуре *ArrayPrint*, не имеют значения. Однако размерность открытых массивов (количество индексов) всегда равна 1 — за этим следит компилятор. Если бы, например, мы добавили в программу двумерный массив *C*

```
var
    C: array [1..3,1..5] of Integer;
```

то обращение

```
ArrayPrint(C)
```

вызвало бы сообщение об ошибке

```
Error26: Type mismatch.
(Ошибка 26: Несоответствие типов.)
```

8.4. ПРОЦЕДУРНЫЕ ТИПЫ. ПАРАМЕТРЫ-ФУНКЦИИ И ПАРАМЕТРЫ-ПРОЦЕДУРЫ

Процедурные типы — это нововведение фирмы *Borland* (в стандартном Паскале таких типов нет). Основное назначение этих типов — дать программисту гибкие средства передачи функций и процедур в качестве фактических параметров обращения к другим процедурам и функциям.

Для объявления процедурного типа используется заголовок процедуры (функции), в котором опускается ее имя, например:

```
type
    Proc1 = Procedure (a, b, c: real; var d: real);
    Proc2 = Procedure (var a, b);
    Proc3 = Procedure;
    Func1 = Function: String;
    Func2 = Function (var s: String): real;
```

Как видно из приведенных примеров, существует два процедурных типа: тип — процедура и тип — функция.

Пример 8.3 иллюстрирует механизм передачи процедур в качестве фактических параметров вызова. Программа выводит на экран таблицу двух функций: $\sin 1(x) = (\sin(x) + 1) \cdot \exp(-x)$ и $\cos 1(x) = (\cos(x) + 1) \cdot \exp(-x)$. Вычисление и печать значений этих функций реализуются в процедуре *PRINTFUNC*, которой в качестве параметров передаются номер позиции *N* на экране, куда будет выводиться очередной результат (с помощью этого параметра реализуется вывод в две колонки), и имя нужной функции.

Пример 8.3.

```

Uses CRT;
type
  Func = Function (x: real): real;
  {-----}
Procedure PrintFunc(XPos: byte; F:Func);
  {Осуществляет печать функции F (XPos - горизонтальная
  позиция начала вывода)}
const
  np = 20;           {Количество вычислений функций}
var
  x : real;
  i : integer;
begin {PrintFunc}
  for i := 1 to np do
    begin
      x := i * (2 * pi / np);
      GotoXY (XPos, WhereY);
      WriteLn (x:5:3, F(x):18:5)
    end
  end; {PrintFunc}
  {-----}
Function Sin1(x: real): real; far;
begin
  sin1 := (sin(x) + 1) * exp(-x)
end;
  {-----}
Function Cos1(x: real): real; far;
begin
  cos1 := (cos(x) + 1) * exp(-x)
end;
  {-----}
begin {основная программа}
  ClrScr;           {Очищаем экран}
  PrintFunc (1, sin1);
  GotoXY (1,1);     {Переводим курсор
                    в левый верхний угол}

  PrintFunc (40, cos1)
end.

```

Обратите внимание: для установления правильных связей функций *SIN1* и *COS1* с процедурой *PRINTFUNC* они должны компилироваться с расчетом на дальнюю модель памяти. Вот почему в программу вставлены стандартные директивы *FAR* сразу за заголовками функций. В таком режиме должны компилироваться любые процедуры (функции), которые будут передаваться в качестве фактических параметров вызова.

Стандартные процедуры (функции) Турбо Паскаля не могут передаваться рассмотренным способом.

В программе могут быть объявлены переменные процедурных типов, например, так:

```
var
  p1      : Proc1;
  f1, f2  : Func2;
  ap      : array [1..N] of Proc1;
```

Переменным процедурных типов допускается присваивать в качестве значений имена соответствующих подпрограмм. После такого присваивания имя переменной становится синонимом имени подпрограммы, например:

```
type
  Proc = Procedure (n: word; var a: byte);
var
  ProcVar: Proc;
  x, y   : byte;
  Procedure Proc1(x: word; var y: byte); far;
begin
  if x > 255 then
    y := x mod 255
  else
    y := byte(x)
end;
begin                                     {Главная программа}
  ProcVar := Proc1;
  for x := 150 to 180 do
    begin
      ProcVar (x + 100, y);
      Write (y:8)
    end
  end.
end.
```

Разумеется, такого рода присваивания допустимы и для параметров — функций, например:

```
type
  FuncType = Function (i : integer) : integer;
var
  VarFunc : FuncType;
  i : integer;
  Function MyFunc (count : integer) : integer; far;
  begin
    .....
  end; {MyFunc}
begin {Основная программа}
  .....
  i := MyFunc(1);   {Обычное использование результата функции}
  .....
```

```
VarFunc := MyFunc; {Присваивание переменной процедурного  
                     типа имени функции MyFunc}
```

```
.....
```

```
end.
```

Отметим, что присваивание

```
VarFunc := MyFunc(1);
```

будет недопустимым, так как слева и справа от знака присваивания используются несовместимые типы: слева — процедурный тип, а справа — *INTEGER*; имя функции со списком фактических параметров *MyFunc(1)* трактуется Турбо Паскалем как обращение к значению функции, в то время как имя функции без списка параметров рассматривается как имя функции.

В отличие от стандартного Паскаля, в Турбо Паскале разрешается использовать в передаваемой процедуре (функции) любые типы параметров: параметры — значения, параметры — переменные, параметры — константы (в стандартном Паскале только параметры — значения).

8.5. НЕТИПИЗИРОВАННЫЕ ПАРАМЕТРЫ—ПЕРЕМЕННЫЕ

Еще одно очень полезное нововведение фирмы Borland — возможность использования *нетипизированных параметров*. Параметр считается нетипизированным, если тип формального параметра — переменной в заголовке подпрограммы не указан, при этом соответствующий ему фактический параметр может быть переменной любого типа. Заметим, что нетипизированными могут быть только параметры — переменные.

Нетипизированные параметры обычно используются в случае, когда тип данных несущественен. Такие ситуации чаще всего возникают при разного рода копированиях одной области памяти в другую, например, с помощью процедур *BLOCKREAD*, *BLOCKWRITE*, *MOVE* и т.п.

Нетипизированные параметры в сочетании с механизмом совмещения данных в памяти (см. п.4.4) можно использовать для передачи подпрограмме одномерных массивов переменной длины (этот способ можно использовать в Турбо Паскале версии 6.0 и более ранней, в которых нет открытых массивов).

В примере 8.4 функция *NORMA* вычисляет норму вектора, длина которого меняется случайным образом. Стандартная константа *MAXINT* содержит максимальное значение целого типа *INTEGER* и равна 32767.

Следует учесть, что при обращении к функции *NORMA* массив *X* помещается в стек и передается по ссылке, поэтому описание локальной переменной *A* в виде одномерного массива максимально возможной длины в 65532 байта (встроенная константа *MAXINT* определяет максимально возможное значение типа *INTEGER* и равна 32767), совпадающего с *X*, на самом деле не приведет к выделению дополнительного объема памяти под размещение этой переменной. Иными словами, переменная *A* — фиктивная переменная, размер которой никак не влияет на объем используемой памяти. С таким же успехом можно было бы объявить ее в виде

массива из одного элемента, правда, в этом случае необходимо позаботиться об отключении контроля выхода индекса за границы диапазона.

Пример 8.4

```

const
  NN = 100; {Максимальная длина вектора}
var
  a      : array [1..NN] of real;
  i, j, N : integer;
  {-----}
Function Norma (var x; N: integer): real;
var
  a : array [1..2*MaxInt div SizeOf(real)] of real absolute x;
  i : integer;
  s : real;
begin {Norma}
  s := 0;
  for i := 1 to N do
    s := s + sqr(a[i]);
  Norma := sqrt(s)
end {Norma};
  {-----}
begin {main}
  for i := 1 to 10 do
    begin
      N := Random(NN) + 1; {Текущая длина вектора}
      for j := 1 to N do
        a[j] := Random;
      WriteLn ('N = ', N:2, '  норма =', Norma(a, N):10:7)
    end
  end {main}.

```

Как видно из рассмотренного примера, передача одномерных массивов переменной длины не вызывает никаких трудностей. Сложнее обстоит дело с многомерными массивами, однако и в этом случае использование описанного приема (нетипизированный параметр и совмещение его в памяти с фиктивной переменной) все-таки проще, чем описанная в гл. 6 индексная арифметика. Еще раз напомним, что в случае многомерных массивов их элементы располагаются в памяти так, что при переходе от младших адресов к старшим наиболее быстро меняется самый правый индекс массива.

8.6. РЕКУРСИЯ И ОПЕРЕЖАЮЩЕЕ ОПИСАНИЕ

Рекурсия — это такой способ организации вычислительного процесса, при котором подпрограмма в ходе выполнения составляющих ее операторов обращается сама к себе.

Рассмотрим классический пример — вычисление факториала (пример 18). Программа вводит с клавиатуры целое число N и выводит на экран

значение $N!$, которое вычисляется с помощью рекурсивной функции *FAC*. Для выхода из программы необходимо либо ввести достаточно большое целое число, чтобы вызвать переполнение при умножении чисел с плавающей запятой, либо нажать *Ctrl-Z* и *Enter*.

При выполнении правильно организованной рекурсивной подпрограммы осуществляется многократный переход от некоторого текущего уровня организации алгоритма к нижнему уровню последовательно до тех пор, пока, наконец, не будет получено тривиальное решение поставленной задачи. В примере 8.5 решение при $N = 0$ тривиально и используется для остановки рекурсии.

Пример 8.5

Program Factorial;

{ $\$S+$ } {Включаем контроль переполнения стека}

var

n: integer;

Function Fac(n: integer): real;

{Рекурсивная функция, вычисляющая $n!$ }

begin {Fac}

if n < 0 **then**

WriteLn ('Ошибка в задании N')

else

if n = 0 **then**

Fac := 1

else Fac := n * Fac(n-1)

end {Fac};

{-----}

begin {main}

repeat

ReadLn(n);

WriteLn('n! = ', Fac(n))

until EOF

end {main}.

Рекурсивная форма организации алгоритма обычно выглядит изящнее итерационной и дает более компактный текст программы, но при выполнении, как правило, медленнее и может вызвать переполнение стека (при каждом входе в подпрограмму ее локальные переменные размещаются в особым образом организованной области памяти, называемой программным стеком). Переполнение стека особенно ощутимо сказывается при работе с сопроцессором: если программа использует арифметический сопроцессор, результат любой вещественной функции возвращается через аппаратный стек сопроцессора, рассчитанный всего на 8 уровней. Если, например, попытаться заменить тип *REAL* функции *FAC* (см. пример 8.5) на *EXTENDED*, программа перестанет работать уже при $N = 8$. Чтобы избежать переполнения стека сопроцессора, следует размещать промежуточные результаты во вспомогательной переменной. Вот правильный вариант примера 8.5 для работы с типом *EXTENDED*:

```

Program Factorial;
{$S+,N+,E+} {Включаем контроль стека и работу сопроцессора}
var
    n: integer;
Function Fac(n: integer): extended;
var
    F: extended;           {Буферная переменная
                           для разгрузки стека сопроцессора}
{Рекурсивная функция, вычисляющая n!}
begin {Fac}
    if n < 0 then
        WriteLn ('Ошибка в задании N')
    else
        if n = 0 then
            Fac := 1
        else
            begin
                F := Fac(n-1);
                Fac := F * n
            end
    end {Fac};
{-----}
begin {main}
    repeat
        ReadLn(n);
        WriteLn('n! = ', Fac(n))
    until EOF
end {main}.

```

Рекурсивный вызов может быть косвенным. В этом случае подпрограмма обращается к себе опосредованно, путем вызова другой подпрограммы, в которой содержится обращение к первой, например:

```

Procedure A (i : byte);
begin
    ...
    B (i);
    ...
end;
Procedure B (j : Byte);
    ...
begin
    ...
    A(j);
    ...
end;

```

Если строго следовать правилу, согласно которому каждый идентификатор перед употреблением должен быть описан, то такую программную

конструкцию использовать нельзя. Для того, чтобы такого рода вызовы стали возможны, вводится *опережающее описание*:

```

Procedure B (j : byte); forward;
Procedure A (i : byte);
begin
    ...
    B (i);
    ...
end;
Procedure B;
begin
    ...
    A(j);
    ...
end;

```

Как видим, опережающее описание заключается в том, что объявляется лишь заголовок процедуры *B*, а ее тело заменяется стандартной директивой **FORWARD**. Теперь в процедуре *A* можно использовать обращение к процедуре *B* — ведь она уже описана, точнее, известны ее формальные параметры, и компилятор может правильным образом организовать ее вызов. Обратите внимание: тело процедуры *B* начинается заголовком, в котором уже не указываются описанные ранее формальные параметры.

8.7. РАСШИРЕННЫЙ СИНТАКСИС ВЫЗОВА ФУНКЦИЙ

В Турбо Паскале есть возможность вызывать функцию и не использовать то значение, которое она возвращает. Иными словами, вызов функции может внешне выглядеть как вызов процедуры, например:

```

{$X+} {Включаем расширенный синтаксис}
Function MyFunc(var x : integer) : integer;
begin
    if x<0 then x:=0
    else MyFunc := x+10
end; {MyFunc}
var
    i : integer;
begin {main}
    i := 1;
    i := 2*MyFunc(i)-100;      {Стандартный вызов функции}
    MyFunc(i)                  {Расширенный синтаксис вызова}
end. {main}

```

Расширенный синтаксис делает использование функций таким же свободным, как, например, их использование в языке Си, и придает Турбо Паскалю дополнительную гибкость. С помощью расширенного синтаксиса нельзя вызывать стандартные функции. Компиляция с учетом расширенного синтаксиса включается активным состоянием опции **EXTENDED**

SYNTAX диалогового окна *OPTIONS/COMPILER* (см. прил.1) или глобальной директивой компилятора *{X+}*.

Глава 9

МОДУЛИ

Стандартный Паскаль не предусматривает механизмов отдельной компиляции частей программы с последующей их сборкой перед выполнением. Более того, последовательное проведение в жизнь принципа обязательного описания любого объекта перед его использованием делает фактически невозможным разработку разнообразных библиотек прикладных программ. Точнее, такие библиотеки в рамках стандартного Паскаля могут существовать только в виде исходных текстов и программист должен сам включать в программу подчас весьма обширные тексты различных поддерживающих процедур, таких, как процедуры матричной алгебры, численного интегрирования, математической статистики и т.п.

Вполне понятно поэтому стремление разработчиков коммерческих компиляторов Паскаля включать в язык средства, повышающие его модульность. Чаще всего таким средством является разрешение использовать внешние процедуры и функции, тело которых заменяется стандартной директивой **EXTERNAL**. Разработчики Турбо Паскаля пошли в этом направлении еще дальше, включив в язык механизм так называемых модулей.

Модуль — это автономно компилируемая программная единица, включающая в себя различные компоненты раздела описаний (типы, константы, переменные, процедуры и функции) и, возможно, некоторые исполняемые операторы инициализирующей части. По своей организации и характеру использования в программе модули Турбо Паскаля близки к модулям-пакетам (**PACKAGE**) языка программирования Ада. В них так же, как в пакетах Ады, явным образом выделяется некоторая «видимая» интерфейсная часть, в которой сконцентрированы описания глобальных типов, констант и переменных, а также приводятся заголовки глобальных процедур и функций. Появление объектов в интерфейсной части делает их доступными для других модулей и основной программы. Тела процедур и функций располагаются в исполняемой части модуля, которая может быть скрыта от пользователя.

Насколько сильно изменяются свойства языка Паскаль при введении механизма модулей свидетельствует следующее замечание его автора Н.Вирта, сделанное им по поводу более позднего языка Модула-2: «Модули — самая важная черта, отличающая язык Модула-2 от его предшественника Паскаля».

Модули представляют собой прекрасный инструмент для разработки библиотек прикладных программ и мощное средство модульного программирования. Важная особенность модулей заключается в том, что компилятор Турбо Паскаля размещает их программный код в отдельном сегменте памяти. Максимальная длина сегмента не может превышать 64 Кбайта, однако количество одновременно используемых модулей ограничивается лишь доступной памятью, что дает возможность создавать весьма крупные программы.

9.1. СТРУКТУРА МОДУЛЕЙ

Модуль имеет следующую структуру:

```
UNIT <имя>;  
INTERFACE
```

```

<интерфейсная часть>
IMPLEMENTATION
<исполняемая часть>
BEGIN
<иницилирующая часть>
END.

```

Здесь **UNIT** — зарезервированное слово (*единица*); начинает заголовок модуля;

<имя> — имя модуля (правильный идентификатор);

INTERFACE — зарезервированное слово (*интерфейс*); начинает интерфейсную часть модуля;

IMPLEMENTATION — зарезервированное слово (*выполнение*); начинается исполняемую часть;

BEGIN — зарезервированное слово; начинает иницилирующую часть модуля; конструкция **BEGIN** *<иницилирующая часть>* необязательна;

END — зарезервированное слово — признак конца модуля.

Таким образом, модуль состоит из заголовка и трех составных частей, любая из которых может быть пустой.

9.2. ЗАГОЛОВОК МОДУЛЯ И СВЯЗЬ МОДУЛЕЙ ДРУГ С ДРУГОМ

Заголовок модуля состоит из зарезервированного слова **UNIT** и следующего за ним имени модуля. Для правильной работы среды Турбо Паскаля и возможности подключения средств, облегчающих разработку крупных программ (см. п.9.6), это имя должно совпадать с именем дискового файла, в который помещается исходный текст модуля. Если, например, имеем заголовок

```
Unit Global;
```

то исходный текст соответствующего модуля должен размещаться в дисковом файле **GLOBAL.PAS**. Имя модуля служит для его связи с другими модулями и основной программой. Эта связь устанавливается специальным предложением

```
USES <сп.модулей>
```

Здесь **USES** — зарезервированное слово (*использует*);

<сп.модулей> — список модулей, с которыми устанавливается связь; элементами списка являются имена модулей, отделяемые друг от друга запятыми, например:

```
Uses CRT, Graph, Global;
```

Если объявление **USES...** используется, оно должно открывать раздел описаний основной программы. Модули могут использовать другие модули. Предложение **USES** в модулях может следовать либо сразу за зарезервированным словом **INTERFACE**, либо сразу за словом **IMPLEMENTATION**, либо, наконец, и там, и там (т.е. допускаются два предложения **USES**).

9.3. ИНТЕРФЕЙСНАЯ ЧАСТЬ

Интерфейсная часть открывается зарезервированным словом **INTERFACE**. В этой части содержатся объявления всех глобальных объектов модуля (типов,

констант, переменных и подпрограмм), которые должны стать доступными основной программе и/или другим модулям. При объявлении глобальных подпрограмм в интерфейсной части указывается только их заголовок, например:

```
Unit Cmplx;
Interface
type
  complex = record
    re, im : real
  end;
Procedure AddC (x, y : complex; var z : complex);
Procedure MulC (x, y : complex; var z : complex);
```

Если теперь в основной программе написать предложение

```
Uses Cmplx;
```

то в программе станут доступными тип *COMPLEX* и две процедуры — *ADDC* и *MULC* из модуля *CMPLX*.

Отметим, что объявление подпрограмм в интерфейсной части автоматически сопровождается их компиляцией с использованием дальней модели памяти (см. гл.8). Таким образом обеспечивается доступ к подпрограммам из основной программы и других модулей.

Следует учесть, что все константы и переменные, объявленные в интерфейсной части модуля, равно как и глобальные константы и переменные основной программы, помещаются компилятором Турбо Паскаля в общий сегмент данных (максимальная длина сегмента 65536 байт). Порядок появления различных разделов объявлений и их количество может быть произвольным. Если в интерфейсной части объявляются внешние подпрограммы или подпрограммы в машинных кодах (см. гл. 11), их тела (т.е. зарезервированное слово *EXTERNAL*, в первом случае, и машинные коды вместе со словом *INLINE* — во втором) должны следовать сразу за их заголовками в исполняемой части модуля (не в интерфейсной!). В интерфейсной части модулей нельзя использовать опережающее описание.

9.4. ИСПОЛНЯЕМАЯ ЧАСТЬ

Исполняемая часть начинается зарезервированным словом *IMPLEMENTATION* и содержит описания подпрограмм, объявленных в интерфейсной части. В ней могут объявляться локальные для модуля объекты — вспомогательные типы, константы, переменные и блоки, а также метки, если они используются в иницилирующей части.

Описанию подпрограммы, объявленной в интерфейсной части модуля, в исполняемой части должен предшествовать заголовок, в котором можно опускать список формальных переменных (и тип результата для функции), так как они уже описаны в интерфейсной части. Но если заголовок подпрограммы приводится в полном виде, т.е. со списком формальных параметров и объявлением результата, он должен совпадать с заголовком, объявленным в интерфейсной части, например:

```
Unit Cmplx;
Interface
```

```

type
  complex = record
    re, im : real
  end;
Procedure AddC (x, y : complex; var z : complex);
Implementation
Procedure AddC;
begin
  z.re := x.re + y.re;
  z.im := x.im + y.im
end;
end.

```

Локальные переменные и константы, а также все программные коды, порожденные при компиляции модуля, помещаются в общий сегмент памяти.

9.5. ИНИЦИИРУЮЩАЯ ЧАСТЬ

Иницирующая часть завершает модуль. Она может отсутствовать вместе с начинающим ее словом **BEGIN** или быть пустой — тогда за **BEGIN** сразу следует признак конца модуля (слово **END** и следующая за ним точка).

В иницирующей части размещаются исполняемые операторы, содержащие некоторый фрагмент программы. Эти операторы исполняются до передачи управления основной программе и обычно используются для подготовки ее работы. Например, в них могут иницироваться переменные, открываться нужные файлы, устанавливаться связи с другими ПК по коммуникационным каналам и т.п.:

```

Unit FileText;
Interface
Procedure Print(s : string);
Implementation
var
  f: text;
const
  name = 'output.txt';
Procedure Print;
begin
  WriteLn(f, s)
end;
{ Начало иницирующей части: }
begin
  assign(f, name);
  rewrite(f);
{ Конец иницирующей части }
end.

```

Не рекомендуется делать иницирующую часть пустой, лучше ее опустить: пустая часть содержит *пустой оператор*, которому будет передано управление при запуске программы. Это часто вызывает проблемы при разработке оверлейных программ (см. гл.11).

9.6. КОМПИЛЯЦИЯ МОДУЛЕЙ

В среде Турбо Паскаля имеются средства, управляющие способом компиляции модулей и облегчающие разработку крупных программных проектов. В частности, определены три режима компиляции: *COMPILE*, *MAKE* и *BUILD* (см. прил.1). Режимы отличаются только способом связи компилируемого модуля или основной программы с другими модулями, объявленными в предложении *USES*.

При компиляции модуля или основной программы в режиме *COMPILE* все упоминающиеся в предложении *USES* модули должны быть предварительно откомпилированы и результаты компиляции помещены в одноименные файлы с расширением *TPU*. Например, если в программе (модуле) имеется предложение

```
Uses Global;
```

то на диске в каталоге, объявленном опцией *UNIT DIRECTORIES* (см. прил.1), уже должен находиться файл *GLOBAL.TPU*. Файл с расширением *TPU* (от англ. *Turbo Pascal Unit*) создается автоматически в результате компиляции модуля (если основная программа может компилироваться без создания исполняемого *EXE*-файла, то компиляция модуля всегда приводит к созданию *TPU*-файла).

В режиме *MAKE* компилятор проверяет наличие *TPU*-файлов для каждого объявленного модуля. Если какой-либо из файлов не обнаружен, система пытается отыскать одноименный файл с расширением *PAS*, т.е. файл с исходным текстом модуля, и, если искомый файл найден, приступает к его компиляции. Кроме того, в этом режиме система следит за возможными изменениями исходного текста любого используемого модуля. Если в *PAS*-файл (исходный текст модуля) внесены какие-либо изменения, то независимо от того, есть ли уже в каталоге соответствующий *TPU*-файл или нет, система осуществляет его компиляцию перед компиляцией основной программы. Более того, если изменения внесены в интерфейсную часть модуля, то будут перекомпилированы также и все другие модули, обращающиеся к нему. Режим *MAKE*, таким образом, существенно облегчает процесс разработки крупных программ с множеством модулей: программист избавляется от необходимости следить за соответствием существующих *TPU*-файлов их исходному тексту, так как система делает это автоматически.

В режиме *BUILD* существующие *TPU*-файлы игнорируются, и система пытается отыскать (и компилировать) соответствующий *PAS*-файл для каждого объявленного в предложении *USES* модуля. После компиляции в режиме *BUILD* программист может быть уверен в том, что учтены все сделанные им изменения в любом из модулей.

Подключение модулей к основной программе и их возможная компиляция осуществляются в порядке их объявления в предложении *USES*. При переходе к очередному модулю система предварительно отыскивает все модули, на которые он ссылается. Ссылки модулей друг на друга могут образовывать древовидную структуру любой сложности, однако запрещается явное или косвенное обращение модуля к самому себе. Например, недопустимы следующие объявления:

```
Unit A;  
Interface  
Uses B;
```

```
Unit B;  
Interface  
Uses A;
```

```

.....
Implementation
.....
end.

```

```

.....
Implementation
.....
end.

```

Это ограничение можно обойти, если «спрятать» предложение *USES* в исполняемые части зависимых модулей:

```

Unit A;
Interface
.....
Implementation
Uses B;
.....
end.

```

```

Unit B;
Interface
.....
Implementation
Uses A;
.....
end.

```

Дело в том, что Турбо Паскаль разрешает ссылки на частично откомпилированные модули, что приблизительно соответствует опережающему описанию подпрограммы. Если интерфейсные части любых двух модулей независимы (это неперенное условие!), Турбо Паскаль сможет идентифицировать все глобальные идентификаторы в каждом из модулей, после чего откомпилирует тела модулей обычным способом.

9.7. ДОСТУП К ОБЪЯВЛЕННЫМ В МОДУЛЕ ОБЪЕКТАМ

Пусть, например, мы создаем модуль, реализующий арифметику комплексных чисел (такая арифметика ни в стандартном Паскале, ни в Турбо Паскале не предусмотрена). К сожалению, в Турбо Паскале нельзя использовать функции, значения которых имели бы структурированный тип (запись, например), поэтому арифметика комплексных чисел реализуется четырьмя процедурами:

```

UNIT Cmplx;
{-----}
                INTERFACE
{-----}
type
    complex = record
                re, im : real
            end;

Procedure AddC (x, y : complex; var z : complex);
Procedure SubC (x, y : complex; var z : complex);
Procedure MulC (x, y : complex; var z : complex);
Procedure DivC (x, y : complex; var z : complex);
cons
    c : complex = (re : 0.1; im : -1);
{-----}
                IMPLEMENTATION
{-----}
Procedure AddC;
begin
    z.re := x.re + y.re;

```

```

    z.im := x.im + y.im
end {AddC};
Procedure SubC;
begin
    z.re := x.re - y.re;
    z.im := x.im - y.im
end {SubC};
Procedure MulC;
begin
    z.re := x.re * y.re - x.im * y.im;
    z.im := x.re * y.im + x.im * y.re
end {MulC};
Procedure DivC;
var
    zz : real;
begin
    zz := sqr(y.re) + sqr(y.im);
    z.re := (x.re * y.re + x.im * y.im) / zz;
    z.im := (x.re * y.im - x.im * y.re) / zz
end {DivC};
end.

```

Текст этого модуля следует поместить в файл *CMPLX.PAS*. Вы можете его откомпилировать, создав *TPU*-файл, после чего Вашей программе станут доступны процедуры из новой библиотеки. Например, в следующей программе (пример 9.1) осуществляются четыре арифметические операции над парой комплексных чисел.

Пример 9.1

```

Uses Cmplx;
var
    a, b, c : complex;
begin
    a.re := 1;  a.im := 1;
    b.re := 1;  b.im := 2;
    AddC(a, b, c);
    WriteLn('Сложение   : ', c.re:5:1, c.im:5:1, 'i');
    SubC(a, b, c);
    WriteLn('Вычитание  : ', c.re:5:1, c.im:5:1, 'i');
    MulC(a, b, c);
    WriteLn('Умножение  : ', c.re:5:1, c.im:5:1, 'i');
    DivC(a, b, c);
    WriteLn('Деление    : ', c.re:5:1, c.im:5:1, 'i');
end.

```

После объявления **Uses** Cmplx программе стали доступны все объекты, объявленные в интерфейсной части модуля *CMPLX*. При необходимости можно переопределить любой из этих объектов, как это произошло, например, с объявленной в модуле типизированной константой *C*. Переопределение объекта означает, что вновь объявленный объект «закрывает» ранее определенный в модуле одноименный объект. Чтобы получить доступ к «закрытому»

объекту, нужно воспользоваться составным именем: перед именем объекта поставить имя модуля и точку. Например, оператор

```
WriteLn(cmplx.c.re:5:1, cmplx.c.in:5:1,'i');
```

выведет на экран содержимое «закрытой» типизированной константы из предыдущего примера.

9.8. СТАНДАРТНЫЕ МОДУЛИ

В Турбо Паскале имеется восемь стандартных модулей, в которых содержится большое число разнообразных типов, констант, процедур и функций. Этими модулями являются *SYSTEM*, *DOS*, *CRT*, *PRINTER*, *GRAPH*, *OVERLAY*, *TURBO3* и *GRAPH3*. Модули *GRAPH*, *TURBO3* и *GRAPH3* выделены в отдельные *TPU*-файлы, а остальные входят в состав библиотечного файла *TURBO.TPL*. Лишь один модуль *SYSTEM* подключается к любой программе автоматически, все остальные становятся доступны только после указания их имен в списке, следующем за словом *USES*.

Ниже приводится краткая характеристика стандартных модулей. Полное описание входящих в них программных средств приведено в прил.4, а описанию объектно-ориентированной библиотеки Turbo Vision посвящена вся вторая часть книги.

Модуль *SYSTEM*. В него входят все процедуры и функции стандартного Паскаля, а также встроенные процедуры и функции, которые не вошли в другие стандартные модули (например, *INC*, *DEC*, *GETDIR* и т.п.). Как уже отмечалось, модуль *SYSTEM* подключается к любой программе независимо от того, объявлен ли он в предложении *USES* или нет, поэтому его глобальные константы, переменные и подпрограммы считаются *встроенными* в Турбо Паскаль.

Модуль *PRINTER*. Делает доступным вывод текстов на матричный принтер. В нем определяется файловая переменная *LST* типа *TEXT*, которая связывается с логическим устройством *PRN*. После подключения модуля может быть выполнена, например, такая программа:

```
Uses Printer;  
begin  
  writeln (LST, 'Турбо Паскаль')  
end.
```

Модуль *CRT*. В нем сосредоточены процедуры и функции, обеспечивающие управление текстовым режимом работы экрана. С помощью входящих в модуль подпрограмм можно перемещать курсор в произвольную позицию экрана, менять цвет выводимых символов и окружающего их фона, создавать окна. Кроме того, в модуль включены также процедуры «слепого» чтения клавиатуры и управления звуком.

Модуль *GRAPH*. Содержит обширный набор типов, констант, процедур и функций для управления графическим режимом работы экрана. С помощью подпрограмм, входящих в модуль *GRAPH*, можно создавать разнообразные графические изображения и выводить на экран текстовые надписи стандартными или разработанными программистом шрифтами. Подпрограммы модуля *GRAPH* после соответствующей настройки могут поддерживать раз-

личные типы аппаратных графических средств. Настройка на имеющиеся в распоряжении программиста технические средства графики осуществляется специальными программами — драйверами, которые не входят в файл *GRAPH.TPU*, но поставляются вместе с ним.

Модуль *DOS*. В модуле собраны процедуры и функции, открывающие доступ программам к средствам дисковой операционной системы *MS-DOS*.

Модуль *OVERLAY*. Он необходим при разработке громоздких программ с перекрытиями. Как уже говорилось, Турбо Паскаль обеспечивает создание программ, длина которых ограничивается лишь основной оперативной памятью ПК. Операционная система *MS-DOS* оставляет исполняемой программе около 580 Кбайт основной памяти (без учета резидентных программ и самой системы Турбо Паскаль). Память такого размера достаточна для большинства применений, тем не менее использование программ с перекрытиями (см. гл.11) снимает это ограничение.

Два библиотечных модуля *TURBO3* и *GRAPH3* введены для совместимости с ранней версией 3.0 системы Турбо Паскаль.

Глава 10

ОБЪЕКТЫ

В основе того или иного языка программирования лежит некоторая руководящая идея, оказывающая существенное влияние на стиль соответствующих программ.

Исторически первой была идея процедурного структурирования программ, в соответствии с которой программист должен был решить, какие именно процедуры он будет использовать в своей программе, а затем выбрать наилучшие алгоритмы для реализации этих процедур. Появление этой идеи было следствием недостаточной изученности алгоритмической стороны вычислительных процессов, столь характерной для ранних программных разработок (сороковые — пятидесятые годы). Типичным примером процедурно-ориентированного языка является Фортран — первый и все еще один из наиболее популярных языков программирования. Последовательное использование идеи процедурного структурирования программ привело к созданию обширных библиотек программирования, содержащих множество сравнительно небольших процедур, из которых, как из кирпичиков, можно строить «здание» программы.

По мере прогресса в области вычислительной математики акцент в программировании стал смещаться с процедур в сторону организации данных. Оказалось, что эффективная разработка сложных программ нуждается в действенных способах контроля правильности использования данных. Контроль должен осуществляться как на стадии компиляции, так и при прогоне программ, в противном случае, как показала практика, резко возрастают трудности создания крупных программных проектов. Отчетливое осознание этой проблемы привело к созданию Алгола-60, а позже — Паскаля, Модулы-2, Си и множества других языков программирования, имеющих более или менее развитые структуры типов данных. Логическим следствием развития этого направления стал модульный подход к разработке программ, характеризующийся стремлением «спрятать» данные и процедуры внутри модуля.

Начиная с языка Симула-67, в программировании наметился новый подход, который получил название *объектно-ориентированного программирования* (ООП). Его руководящая идея заключается в стремлении связать данные с обрабатывающими эти данные процедурами в единое целое — *объект*. Характерной чертой объектов является *инкапсуляция* (объединение) данных и алгоритмов их обработки, в результате чего и данные, и процедуры во многом теряют самостоятельное значение. Фактически объектно-ориентированное программирование можно рассматривать как модульное программирование нового уровня, когда вместо во многом случайного, механического объединения процедур и данных акцент делается на их смысловую связь.

Какими мощными средствами располагает объектно-ориентированное программирование наглядно демонстрирует библиотека Turbo Vision.

входящая в комплект поставки Турбо Паскаля и описываемая во второй части этой книги. В этой главе мы рассмотрим основные идеи ООП и способы их использования.

Следует заметить, что преимущества ООП в полной мере проявляются лишь при разработке достаточно сложных программ. Более того, инкапсуляция придает объектам совершенно особое свойство «самостоятельности», максимальной независимости от остальных частей программы. Правильно сконструированный объект располагает всеми необходимыми данными и процедурами их обработки, чтобы успешно реализовать требуемые от него действия. Попытки использовать ООП для программирования несложных алгоритмов, связанных, например, с расчетными вычислениями по готовым формулам, чаще всего выглядят искусственными нагромождениями ненужных языковых конструкций. Такие программы обычно не нуждаются в структуризации, расчленении алгоритма на ряд относительно независимых частей, их проще и естественнее разрабатывать традиционными способами Паскаля. При разработке сложных диалоговых программ программист вынужден структурировать программу, так как только в этом случае он может рассчитывать на успех: «критической массой» неструктурированных программ является объем в 1000–1200 строк исходного текста — отладка неструктурированных программ большего объема обычно сталкивается с чрезмерными трудностями. Структурирование программы ведет, фактически, к разработке собственной библиотеки программирования — вот в этот момент к Вам на помощь и приходят новые средства ООП.

10.1. ОСНОВНЫЕ ПРИНЦИПЫ ООП

Объектно — ориентированное программирование основано на «трех китах» — трех важнейших принципах, придающих объектам новые свойства. Этими принципами являются инкапсуляция, наследование и полиморфизм.

Инкапсуляция

Инкапсуляция есть объединение в единое целое данных и алгоритмов обработки этих данных. В рамках ООП данные называются *полями* объекта, а алгоритмы — *объектными методами*.

Инкапсуляция позволяет в максимальной степени изолировать объект от внешнего окружения. Она существенно повышает надежность разрабатываемых программ, т.к. локализованные в объекте алгоритмы обмениваются с программой сравнительно небольшими объемами данных, причем количество и тип этих данных обычно тщательно контролируются. В результате замена или модификация алгоритмов и данных, инкапсулированных в объект, как правило, не влечет за собой плохо прослеживаемых последствий для программы в целом (в целях повышения защищенности программ в ООП почти не используются глобальные переменные).

Другим немаловажным следствием инкапсуляции является легкость обмена объектами, переноса их из одной программы в другую. Можно

сказать, что ООП «провоцирует» разработку библиотек объектов, таких как *Turbo Vision*.

Наследование

Наследование есть свойство объектов порождать своих потомков. Объект — потомок автоматически наследует от родителя все поля и методы, может дополнять объекты новыми полями и заменять (перекрывать) методы родителя или дополнять их.

Принцип наследования решает проблему модификации свойств объекта и придает ООП в целом исключительную гибкость. При работе с объектами программист обычно подбирает объект, наиболее близкий по своим свойствам для решения конкретной задачи, и создает одного или нескольких потомков от него, которые «умеют» делать то, что не реализовано в родителе.

Последовательное проведение в жизнь принципа «наследуй и изменяй» хорошо согласуется с поэтапным подходом к разработке крупных программных проектов и во многом стимулирует такой подход.

Полиморфизм

Полиморфизм — это свойство родственных объектов (т.е. объектов, имеющих одного общего родителя) решать схожие по смыслу проблемы разными способами. В рамках ООП поведенческие свойства объекта определяются набором входящих в него методов. Изменяя алгоритм того или иного метода в потомках объекта, программист может придавать этим потомкам отсутствующие у родителя специфические свойства. Для изменения метода необходимо *перекрыть* его в потомке, т.е. объявить в потомке одноименный метод и реализовать в нем нужные действия. В результате в объекте — родителе и объекте — потомке будут действовать два *одноименных* метода, имеющие разную алгоритмическую основу и, следовательно, придающие объектам разные свойства. Это и называется полиморфизмом объектов.

В Турбо Паскале полиморфизм достигается не только описанным выше механизмом наследования и перекрытия методов родителя, но и их *виртуализацией* (см. ниже), позволяющей родительским методам обращаться к методам потомков.

10.2. ПОСТАНОВКА УЧЕБНОЙ ЗАДАЧИ

Знакомство с техникой ООП в этом разделе иллюстрируется примерами, объединенными рамками следующей учебной задачи.

Требуется разработать программу, которая создает на экране ряд графических изображений (точки, окружность, линия, квадрат) и может перемещать эти изображения по экрану. Вид создаваемого программой экрана показан на рис.10.1.

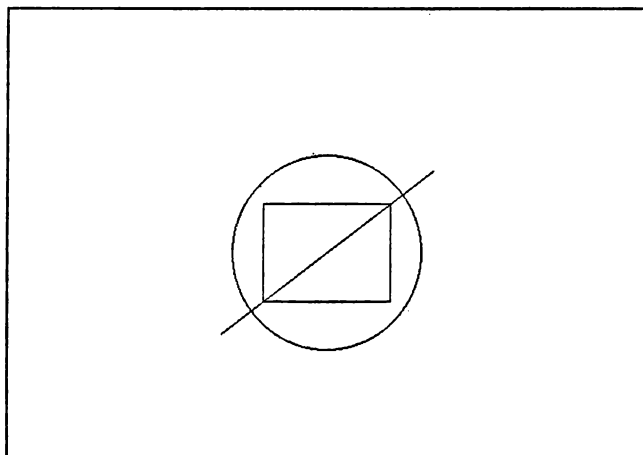


Рис.10.1. Экран, создаваемый учебной программой

Для перемещения изображений в программе будут использоваться клавиши управления курсором, клавиши *Home*, *End*, *PgUp*, *PgDn* (для перемещения по диагональным направлениям) и клавиша *Tab* для выбора перемещаемого объекта. Выход из программы — клавиша *Esc*.

Техническая реализация программы потребует использования средств двух стандартных библиотек — *CRT* и *GRAPH*, которые еще не рассматривались в этой книге. Чтобы не отвлекать Ваше внимание от основных проблем ООП, при описании реализации учебной задачи особенности использования средств этих библиотек лишь очень кратко комментируются в текстах программы. Если Вы не привыкли «принимать на веру» предлагаемые программные решения и хотите разобраться с деталями вызова незнакомых Вам процедур и функций, рекомендую просмотреть материал гл.13 и гл.14, где описаны эти библиотеки (они не используют средств ООП и, следовательно, могут изучаться до чтения настоящей главы).

10.3. СОЗДАНИЕ ОБЪЕКТОВ

В Турбо Паскале для создания объектов используются три зарезервированных слова: ***object***, ***constructor***, ***destructor*** и три стандартные директивы: ***private***, ***public*** и ***virtual***.

Зарезервированное слово ***object*** используется для описания объекта. Описание объекта должно помещаться в разделе описания типов:

```
type
  MyObject = object
    {Поля объекта}
    {Методы объекта}
  end;
```

Если объект порождается от какого-либо родителя, имя родителя указывается в круглых скобках сразу за словом ***object***:

```
type
  MyDescendantObject = object(MyObject)
```

.
.
end;

Любой объект может иметь сколько угодно потомков, но только одного родителя, что позволяет создавать иерархические деревья наследования объектов.

Для нашей учебной задачи создадим объект-родитель *TGraphObject*, в рамках которого будут инкапсулированы поля и методы, общие для всех остальных объектов:

type

```
TGraphObj = object
Private      {Поля объекта будут скрыты от пользователя}
  X,Y: Integer; {Координаты реперной точки}
  Color: Word;  {Цвет фигуры}
Public      {Методы объекта будут доступны пользователю}
  Constructor Init(aX,aY: Integer; aColor: Word);
    {Создает экземпляр объекта}
  Procedure Draw(aColor: Word); Virtual;
    {Вычерчивает объект заданным цветом aColor}
  Procedure Show;
    {Показывает объект - вычерчивает его цветом Color}
  Procedure Hide;
    {Прячет объект - вычерчивает его цветом фона}
  Procedure MoveTo(dX,dY: Integer);
    {Перемещает объект в точку с координатами X+dX и Y+dY}
end; {Конец описания объекта TGraphObj}
```

В дальнейшем предполагается создать объекты-потомки от *TGraphObj*, реализующие все специфические свойства точки, линии, окружности и прямоугольника. Каждый из этих графических объектов будет характеризоваться положением на экране (поля *X* и *Y*) и цветом (поле *Color*). С помощью метода *Draw* он будет способен отображать себя на экране, а с помощью свойств «показать себя» (метод *Show*) и «спрятать себя» (метод *Hide*) сможет перемещаться по экрану (метод *MoveTo*). Учитывая общность свойств графических объектов, мы объявляем абстрактный объект *TGraphObj*, который не связан с конкретной графической фигурой. Он объединяет в себе все общие поля и методы реальных фигур и будет служить родителем для других объектов.

Директива *Private* в описании объекта открывает секцию описания скрытых полей и методов. Перечисленные в этой секции элементы объекта «не видны» программисту, если этот объект он получил в рамках библиотечного *TPU*-модуля. Скрываются обычно те поля и методы, к которым программист (в его же интересах!) не должен иметь непосредственного доступа. В нашем примере он не может произвольно менять координаты реперной точки (*X,Y*), т.к. это не приведет к перемещению объекта. Для изменения полей *X* и *Y* предусмотрены входящие в состав объекта методы *Init* и *MoveTo*. Скрытые поля и методы доступны в рамках

той программной единицы (программы или модуля), где описан соответствующий объект. В дальнейшем предполагается, что программа будет использовать модуль *GraphObj* с описанием объектов. Скрытые поля будут доступны в модуле *GraphObj*, но недоступны в использующей его основной программе. Разумеется, в рамках реальной задачи создание скрытых элементов объекта вовсе необязательно. Я ввел их в объект *TGraphObj* лишь для иллюстрации возможностей ООП.

Директива *public* отменяет действие директивы *private*, поэтому все следующие за *public* элементы объекта доступны в любой программной единице. Директивы *private* и *public* могут произвольным образом чередоваться в пределах одного объекта.

Вариант объявления объекта *TGraphObj* без использования механизма *private...public*:

type

```
TGraphObj = object
  X,Y: Integer;
  Color: Word;
  Constructor Init(aX,aY: Integer; aColor: Word);
  Procedure Draw(aColor: Word); Virtual;
  Procedure Show;
  Procedure Hide;
  Procedure MoveTo(dX,dY: Integer);
```

end;

Описания полей ничем не отличаются от описания обычных переменных. Полями могут быть любые структуры данных, в том числе и другие объекты. Используемые в нашем примере поля *X* и *Y* содержат координату реперной (характерной) точки графического объекта, а поле *Color* — его цвет. Реперная точка характеризует текущее положение графической фигуры на экране и, в принципе, может быть любой ее точкой¹.

Для описания методов в ООП используются традиционные для Паскаля процедуры и функции, а также особый вид процедур — *конструкторы* и *деструкторы*. Конструкторы предназначены для создания конкретного экземпляра объекта, ведь объект — это тип данных, т.е. «шаблон», по которому можно создать сколько угодно рабочих экземпляров данных объектного типа (типа *TGraphObj*, например). Зарезервированное слово *constructor*, используемое в заголовке конструктора вместо *procedure*, предписывает компилятору создать особый код пролога, с помощью которого настраивается так называемая таблица виртуальных методов (см. ниже). Если в объекте нет виртуальных методов, в нем может не быть ни одного конструктора, наоборот, если хотя бы один метод описан как виртуальный (с последующим словом *Virtual*, см. метод *Draw*), в состав

¹ В нашем примере она совпадает с координатами точки в описываемом ниже объекте *TPoint*, с центром окружности в объекте *TCircle*, первым концом прямой в объекте *TLine* и с левым верхним углом прямоугольника в объекте *TRect*.

объекта должен входить хотя бы один конструктор и обращение к конструктору должно предшествовать обращению к любому виртуальному методу.

Типичное действие, реализуемое конструктором, состоит в наполнении объектных полей конкретными значениями. Следует заметить, что разные экземпляры одного и того же объекта отличаются друг от друга только содержимым объектных полей, в то время как каждый из них использует одни и те же объектные методы. В нашем примере конструктор *Init* объекта *TGraphObj* получает все необходимые для полного определения экземпляра данные через параметры обращения *aX*, *aY* и *aColor*.

Процедура *Draw* предназначена для вычерчивания графического объекта. Эта процедура будет реализовываться в потомках объекта *TGraphObj* по-разному. Например, для визуализации точки следует вызвать процедуру *PutPixel*, для вычерчивания линии — процедуру *Line* и т.д. В объекте *TGraphObj* процедура *Draw* определена как виртуальная («воображаемая»). Абстрактный объект *TGraphObj* не предназначен для вывода на экран, однако наличие процедуры *Draw* в этом объекте говорит о том, что любой потомок *TGraphObj* должен иметь собственный метод *Draw*, с помощью которого он может показать себя на экране.

При трансляции объекта, содержащего виртуальные методы, создается так называемая *таблица виртуальных методов* (TBM), количество элементов которой равно количеству виртуальных методов объекта. В этой таблице будут храниться адреса точек входа в каждый виртуальный метод. В нашем примере TBM объекта *TGraphObj* хранит единственный элемент — адрес метода *Draw*. Первоначально элементы TBM не содержат конкретных адресов. Если бы мы создали экземпляр объекта *TGraphObj* с помощью вызова его конструктора *Init*, код пролога конструктора поместил бы в TBM нужный адрес родительского метода *Draw*. Далее мы создадим несколько потомков объекта *TGraphObj*. Каждый из них будет иметь собственный конструктор, с помощью которого TBM каждого потомка настраивается так, чтобы ее единственный элемент содержал адрес нужного метода *Draw*. Такая процедура называется *поздним связыванием* объекта. Позднее связывание позволяет методам родителя обращаться к виртуальным методам своих потомков и использовать их для реализации специфичных для потомков действий.

Наличие в объекте *TGraphObj* виртуального метода *Draw* позволяет легко реализовать три других метода объекта: чтобы показать объект на экране в методе *Show*, вызывается *Draw* с цветом *aColor*, равным значению поля *Color*, а чтобы спрятать графический объект, в методе *Hide* вызывается *Draw* со значением цвета *GetBkColor*, т.е. с текущим цветом фона.

Рассмотрим реализацию перемещения объекта. Если потомок *TGraphObj* (например, *TLine*) хочет переместить себя на экране, он обращается к родительскому методу *MoveTo*. В этом методе сначала с помощью *Hide* объект стирается с экрана, а затем с помощью *Show* показывается в другом месте. Для реализации своих действий и *Hide*, и *Show* обращаются к виртуальному методу *Draw*. Поскольку вызов *MoveTo* происходит в рамках объекта *TLine*, используется TBM этого объекта и вызывается его метод *Draw*, вычерчивающий прямую. Если бы перемещалась окружность, TBM

содержала бы адрес метода *Draw* объекта *TCircle* и визуализация — стирание объекта осуществлялась бы с помощью этого метода.

Чтобы описать все свойства объекта, необходимо раскрыть содержимое объектных методов, т.е. описать соответствующие процедуры и функции. Описание методов производится обычным для Паскаля способом в любом месте раздела описаний, но после описания объекта. Например:

type

 TGraphObj = **object**

 .

 .

end;

Constructor TGraphObj.Init;

begin

 X := aX;

 Y := aY;

 Color := aColor

end;

Procedure TGraphObj.Draw;

begin

{Эта процедура в родительском объекте ничего не делает, поэтому экземпляры TGraphObj не способны отображать себя на экране. Чтобы потомки объекта TGraphObj были способны отображать себя, они должны перекрывать этот метод}

end;

Procedure TGraphObj.Show;

begin

 Draw(Color)

end;

Procedure TGraphObj.Hide;

begin

 Draw(GetBkColor)

end;

Procedure TGraphObj.MoveTo;

begin

 Hide;

 X := X+dX;

 Y := Y+dY;

 Show

end;

Отмечу два обстоятельства. Во-первых, при описании методов имя метода дополняется спереди именем объекта, т.е. используется *составное* имя метода. Это необходимо по той простой причине, что в иерархии

родственных объектов любой из методов может быть перекрыт в потомках. Составные имена четко указывают принадлежность конкретной процедуры. Во-вторых, в любом объектном методе можно использовать инкапсулированные поля объекта почти так, как если бы они были определены в качестве глобальных переменных. Например, в конструкторе *TGraph.Init* переменные в левых частях операторов присваивания представляют собой объектные поля и не должны заново описываться в процедуре. Более того, описание

```

Constructor TGraphObj.Init;
var
    X,Y: Integer;           {Ошибка!}
    Color: Word;            {Ошибка!}
begin
    .
    .
end;

```

вызовет сообщение о двойном определении переменных *X*, *Y* и *Color* (в этом и состоит отличие в использовании полей от глобальных переменных: глобальные переменные можно переопределять в процедурах, в то время как объектные поля переопределять нельзя).

Обратите внимание: абстрактный объект *TGraphObj* не предназначен для вывода на экран, поэтому его метод *Draw* ничего не делает. Однако методы *Hide*, *Show* и *MoveTo* «знают» формат вызова этого метода и реализуют необходимые действия, обращаясь к реальным методам *Draw* своих будущих потомков через соответствующие ТВМ. Это и есть полиморфизм объектов.

Создадим простейшего потомка от *TGraphObj* — объект *TPoint*, с помощью которого будет визуализироваться и перемещаться точка. Все основные действия, необходимые для этого, уже есть в объекте *TGraphObj*, поэтому в объекте *TPoint* перекрывается единственный метод — *Draw*:

```

type
    TPoint = object (TGraphObj)
        Procedure Draw(aColor); Virtual;
    end;

Procedure TPoint.Draw;
begin
    PutPixel(X,Y,Color) {Показываем цветом Color пиксель
                        с координатами X и Y}
end;

```

В новом объекте *TPoint* можно использовать любые методы объекта-родителя *TGraphObj*. Например, вызвать метод *MoveTo*, чтобы переместить изображение точки на новое место. В этом случае родительский метод *TGraphObj.MoveTo* будет обращаться к методу *TPoint.Draw*, чтобы спрятать и затем показать изображение точки. Такой вызов станет доступен после обращения к конструктору *Init* объекта *TPoint*, который нужным образом

настроит TBM объекта. Если вызвать *TPoint.Draw* до вызова *Init*, его TBM не будет содержать правильного адреса и программа «зависнет».

Чтобы создать объект—линию, необходимо ввести два новых поля для хранения координат второго конца. Дополнительные поля требуется наполнить конкретными значениями, поэтому нужно перекрыть конструктор родительского объекта:

type

```
TLine = object(TGraphObj)
  dX,dY: Integer;      {Приращения координат второго конца}
  Constructor Init(X1,Y1,X2,Y2: Integer; aColor: Word);
  Procedure Draw(aColor: Word); Virtual;
end;
```

Constructor TLine.Init;

{Вызывает унаследованный конструктор TGraphObj для инициации полей X, Y и Color. Затем иницирует поля dX и dY}

begin

{Вызываем унаследованный конструктор}

Inherited Init(X1,Y1,aColor);

{Инициуем поля dX и dY}

dX := X2-X1;

dY := Y2-Y1

end;

Procedure Draw;

begin

SetColor(Color); {Устанавливаем цвет Color}

Line(X,Y,X+dX,Y+dY) {Вычерчиваем линию}

end;

В конструкторе *TLine.Init* для инициации полей X, Y и Color, унаследованных от родительского объекта, вызывается унаследованный конструктор *TGraph.Init*, для чего используется зарезервированное слово *inherited* (англ. — унаследованный):

```
Inherited Init(X1,Y1,aColor);
```

С таким же успехом мы могли бы использовать и составное имя метода:

```
TGraphObj.Init(X1,Y1,aColor);
```

Для инициации полей *dX* и *dY* вычисляется расстояние в пикселах по горизонтали и вертикали от первого конца прямой до ее второго конца. Это позволяет в методе *TLine.Draw* вычислить координаты второго конца по координатам первого и смещениям *dX* и *dY*. В результате простое изменение координат реперной точки X,Y в родительском методе *TGraph.MoveTo* перемещает всю фигуру по экрану.

Теперь нетрудно реализовать объект *TCircle* для создания и перемещения окружности:

```

type
  TCircle = object(TGraphObj)
    R: Integer;           {Радиус}
    Constructor Init(aX,aY,aR: Integer; aColor: Word);
    Procedure Draw(aColor: Virtual);
  end;

Constructor TCircle.Init;
begin
  Inherited Init(aX,aY,aColor);
  R := aR
end;

Procedure TCircle.Draw;
begin
  SetColor(aColor);      {Устанавливаем цвет Color}
  Circle(X,Y,R)           {Вычерчиваем окружность}
end;

```

В объекте *TRect*, с помощью которого создается и перемещается прямоугольник, учтем то обстоятельство, что для задания прямоугольника требуется указать четыре целочисленных параметра, т.е. столько же, сколько для задания линии. Поэтому объект *TRect* удобнее породить не от *TGraphObj*, а от *TLine*, чтобы использовать его конструктор *Init*:

```

type
  TRect = object(TLine)
    Procedure Draw(aColor: Word);
  end;

Procedure TRect.Draw;
begin
  SetColor(aColor);
  Rectangle(X,Y,X+dX,Y+dY) {Вычерчиваем прямоугольник}
end;

```

Чтобы описания графических объектов не мешали созданию основной программы, оформим эти описания в отдельном модуле *GraphObj*:

```

Unit GraphObj;
Interface
{Интерфейсная часть модуля содержит только
объявления объектов}
type
  TGraphObj = object
    .
    .
  end;
  TPoint = object(TGraphObj)

```

```

    .
    .
end;
TLine = object(TGraphObj)
    .
    .
end;
TCircle = object(TGraphObj)
    .
    .
end;
TRect = object(TLine)
    .
    .
end;
Implementation
{Исполняемая часть содержит описания всех объектных методов}
Uses Graph;
Constructor TGraphObj.Init;
    .
    .
end.

```

В интерфейсной части модуля приводятся лишь объявления объектов, подобно тому как описываются другие типы данных, объявляемые в модуле доступными для внешних программных единиц. Расшифровка объектных методов помещается в исполняемую часть *implementation*, как если бы это были описания обычных интерфейсных процедур и функций. При описании методов можно опускать повторное описание в заголовке параметров вызова. Если они все же повторяются, они должны в точности соответствовать ранее объявленным параметрам в описании объекта. Например, заголовок конструктора *TGraphObj.Init* может быть таким:

Constructor TGraphObj.Init;

или таким:

Constructor TGraphObj.Init(aX, aY: Integer; aColor: Word);

10.4. ИСПОЛЬЗОВАНИЕ ОБЪЕКТОВ

Идею инкапсуляции полей и алгоритмов можно применить не только к графическим объектам, но и ко всей программе в целом. Ничто не мешает нам создать объект — программу и «научить» его трем основным действиям: инициации (*Init*), выполнению основной работы (*Run*) и завершению (*Done*). На этапе инициации экран переводится в графический режим работы и создаются и отображаются графические объекты (100 экземпляров *TPoint* и по одному экземпляру *TLine*, *TCircle*, *TRect*). На этапе *Run* осуществляется сканирование клавиатуры и перемещение графических

объектов. Наконец, на этапе *Done* экран переводится в текстовый режим и завершается работа всей программы.

Назовем объект—программу именем *TGraphApp* и разместим его в модуле *GraphApp* (пока не обращайтесь внимание на точки, скрывающие содержательную часть модуля — позднее будет представлен его полный текст):

```
Unit GraphApp;
Interface
type
  TGraphApp = object
    Procedure Init;
    Procedure Run;
    Destructor Done;
  end;
Implementation
Procedure TGraphApp.Init;
.
.
end;
.
.
end.
```

В этом случае основная программа будет предельно простой:

```
Program Graph_Objects;
Uses GraphApp;
var
  App: TGraphApp;
begin
  App.Init;
  App.Run;
  App.Done
end.
```

В ней мы создаем единственный экземпляр *App* объекта—программы *TGraphApp* и обращаемся к трем его методам.

Создание экземпляра объекта ничуть не отличается от создания экземпляра переменной любого другого типа. Просто в разделе описания переменных мы указываем имя переменной и ее тип:

```
var
  App: TGraphApp;
```

Получив это указание, компилятор зарезервирует нужный объем памяти для размещения всех полей объекта *TGraphApp*. Чтобы обратиться к тому или иному объектному методу или полю, используется составное имя, причем первым указывается не имя объектного типа, а имя соответствующей переменной:

```

App.Init;
App.Run;
App.Done;

```

Переменные объектного типа могут быть статическими или динамическими, т.е. располагаться в сегменте данных (статические) или в куче (динамические). В последнем случае мы могли бы использовать такую программу:

```

Program Graph_Objects;
Uses GraphApp;
type
    PGraphApp = ^TGraphApp;
var
    App: PGraphApp;
begin
    App := New(PGraphApp, Init)
    App^.Run;
    App^.Done
end;

```

Для инициации динамической переменной *App* используется вызов функции *New*. В этом случае первым параметром указывается имя типа иницируемой переменной, а вторым осуществляется вызов метода-конструктора, который, я напомним, нужен для настройки таблицы виртуальных методов. Такой прием (распределение объектов в динамической памяти с одновременной инициацией их TBM) характерен для техники ООП.

Ниже приводится возможный вариант модуля *GraphApp* для нашей учебной программы:

```
Unit GraphApp;
```

```
Interface
```

```
Uses GraphObj;
```

```
const
```

```
NPoints = 100;
```

```
{Количество точек}
```

```
type
```

```
{Объект-программа}
```

```
TGraphApp = object
```

```
    Points: array [1..NPoints] of TPoint; {Массив точек}
```

```
    Line: TLine; {Линия}
```

```
    Rect: TRect; {Прямоугольник}
```

```
    Circle: TCircle; {Окружность}
```

```
    ActiveObj: Integer; {Активный объект}
```

```
    Procedure Init;
```

```
    Procedure Run;
```

```
    Procedure Done;
```



```

Procedure ShowAll;
Procedure MoveActiveObj (dX,dY: Integer);
end;

```

Implementation

Uses Graph,CRT;

```

Procedure TGraphApp.Init;

```

{Инициализирует графический режим работы экрана. Создает и отображает NPoints экземпляров объекта TPoint, а также экземпляры объектов TLine, TCircle и TRect}

```

var

```

```

    D,R,Err,k: Integer;

```

```

begin

```

```

    {Инициализируем графику}

```

```

    D := Detect; {Режим автоматического определения
                  типа графического адаптера}

```

```

    InitGraph(D,R,'\\tp\\bgi'); {Инициализируем графический режим.
    Текстовая строка должна задавать путь к каталогу с графическими
    драйверами}

```

```

    Err := GraphResult; {Проверяем успех инициации графики}

```

```

    if Err<>0 then

```

```

        begin

```

```

            GraphErrorMsg(Err);

```

```

            Halt

```

```

        end;

```

```

    {Создаем точки}

```

```

    for k := 1 to NPoints do

```

```

Points[k].Init(Random(GetMaxX),Random(GetMaxY),Random(15)+1);

```

```

    {Создаем другие объекты}

```

```

    Line.Init(GetMaxX div 3,GetMaxY div 3,2*GetMaxX div 3,
              2*GetMaxY div 3,LightRed);

```

```

    Circ.Init(GetMaxX div 2,GetMaxY div 2,GetMaxY div 5,White);

```

```

    Rect.Init(2*GetMaxX div 5,2*GetMaxY div 5,3*GetMaxX div 5,
              3*GetMaxY div 5,Yellow);

```

```

    ShowAll; {Показываем все графические объекты}

```

```

    ActiveObj := 1 {Первым перемещаем прямоугольник}

```

```

end; {TGraphApp.Init}

```

```

{-----}

```

```

Procedure TGraphApp.Run;

```

{Выбирает объект с помощью Tab и перемещает его по экрану}

```

var

```

```

    Stop: Boolean;

```

```

    {Признак нажатия Esc}

```

```

const

```

```

    D = 5;

```

```

    {Шаг смещения фигур}

```

```

begin
  Stop := False;
  {Цикл опроса клавиатуры}
  repeat
    case ReadKey of      {Читаем код нажатой клавиши}
      #27: Stop := True;      {Нажата Esc}
      #9: begin              {Нажата Tab}
        inc(ActiveObj);
        if ActiveObj > 3 then
          ActiveObj := 3
        end;
      #0: case ReadKey of
        #71: MoveActiveObj(-D, -D); {Влево и вверх}
        #72: MoveActiveObj( 0, -D); {Вверх}
        #73: MoveActiveObj( D, -D); {Вправо и вверх}
        #75: MoveActiveObj(-D, 0); {Влево}
        #77: MoveActiveObj( D, 0); {Вправо}
        #79: MoveActiveObj(-D, D); {Влево и вниз}
        #80: MoveActiveObj( 0, D); {Вниз}
        #81: MoveActiveObj( D, D); {Вправо и вниз}
      end
    end;
  ShowAll;
  Until Stop
end; {TGraphApp.Run}
{-----}
Destructor TGraphApp.Done;
{Закрывает графический режим}
begin
  CloseGraph
end; {TGraphApp.Done}
{-----}
Procedure TGraphApp.ShowAll;
{Показывает все графические объекты}
var
  k: Integer;
begin
  for k := 1 to NPoints do Points[k].Show;
  Line.Show;
  Rect.Show;
  Circ.Show
end;
{-----}
Procedure TGraphApp.MoveActiveObj;
{Перемещает активный графический объект}
begin
  case ActiveObj of

```

```
1: Rect.MoveTo (dX, dY);  
2: Circ.MoveTo (dX, dY);  
3: Line.MoveTo (dX, dY)  
end  
end;  
end.
```

В реализации объекта *TGraphApp* используется деструктор *Done*. Следует иметь в виду, что в отличие от конструктора, осуществляющего настройку ТВМ, деструктор не связан с какими-то специфичными действиями: для компилятора слова *destructor* и *procedure* — синонимы. Введение в ООП деструкторов носит, в основном, стилистическую направленность — просто процедуру, разрушающую экземпляр объекта, принято называть деструктором. В реальной практике ООП с деструкторами обычно связывают процедуры, которые не только прекращают работу с объектом, но и освобождают выделенную для него динамическую память. И хотя в нашем примере деструктор *Done* не освобождает кучу, я решил использовать общепринятую стилистику и заодно обсудить с Вами последнее еще не рассмотренное зарезервированное слово технологии ООП.

В заключении следует сказать, что формалистика ООП в рамках реализации этой технологии в Турбо Паскале предельно проста и лаконична. Согласитесь, что введение лишь шести зарезервированных слов, из которых действительно необходимыми являются три (*object*, *constructor* и *virtual*), весьма небольшая плата за мощный инструмент создания современного программного обеспечения.

ДРУГИЕ ВОЗМОЖНОСТИ ТУРБО ПАСКАЛЯ

11.1. ВНЕШНИЕ ПРОЦЕДУРЫ (ФУНКЦИИ)

С помощью внешних процедур (функций) можно осуществить вызов из программы процедур или функций, написанных на языке ассемблера. Ассемблер обеспечивает компиляцию программ, написанных на машинно-ориентированном языке программирования низкого уровня. В Турбо Паскале есть собственный встроенный ассемблер (см. гл.12). В этом разделе речь идет о программах, написанных и откомпилированных с помощью внешнего ассемблера, такого, как, например, ассемблер фирмы *MicroSoft* или *Turbo Assembler* фирмы *Borland*.

Машинно-ориентированный язык ассемблера предоставляет квалифицированному программисту богатейшие возможности использования всех особенностей архитектуры ПК. Ассемблерные программы выполняются значительно быстрее и занимают меньший объем памяти, чем программы, написанные на Турбо Паскале, однако низкий уровень языка ассемблера существенно снижает производительность труда программиста и резко усложняет отладку программ. Как правило, на языке ассемблера пишутся сравнительно небольшие фрагменты программ, в которых используются недоступные из Турбо Паскаля особенности архитектуры ПК.

Внешняя процедура (функция) в программе, написанной на Турбо Паскале, объявляется своим заголовком, за которым следует стандартная директива **EXTERNAL**, например:

```
Function LoCase (ch : char) : char; external;  
Procedure Swapping (var a,b; N : word); external;
```

Как видно из этих примеров, тело внешней процедуры (функции) отсутствует — его заменяет директива **EXTERNAL**. Для подключения ассемблерной программы необходимо предварительно ее откомпилировать и получить объектный файл с расширением **.OBJ**, содержащий перемещаемый код ассемблерной программы. Непосредственно перед описанием внешней процедуры (функции) в тело основной программы вставляется директива компилятора **{ \$L <имя файла> }**, где **<имя файла>** — имя **OBJ**-файла. Диск и каталог, в котором следует искать этот файл, если он не обнаружен в текущем каталоге, указываются опцией **OPTIONS/DIRECTORIES/OBJECT DIRECTORIES** (см. прил.1).

Перед передачей управления внешней процедуре (функции) программа помещает параметры обращения в программный стек в том порядке, как они перечислены в заголовке процедуры (функции). Ассемблерная процедура должна сохранить регистры **BP**, **SP**, **SS** и **DS** центрального процессора в самом начале своей работы и восстановить содержимое этих

регистров перед возвратом управления в программу. Остальные регистры можно не сохранять и соответственно не восстанавливать.

Параметры могут передаваться по ссылке или по значению. Если параметр передается по ссылке, в стек помещается указатель, содержащий абсолютный адрес параметра, если по значению — в стек помещается сам параметр, точнее — его значение. Все параметры — переменные, т.е. параметры, объявленные в заголовке с предшествующим словом *VAR*, всегда передаются по ссылке. Параметры — значения могут передаваться по ссылке или по значению в зависимости от длины внутреннего представления соответствующего параметра. В общем случае используется следующее правило: если длина внутреннего представления параметра — значения составляет 1, 2 или 4 байта, он передается своим значением, т.е. его значение помещается в стек. Точно так же через стек передаются и все вещественные данные длиной в 4, 6, 8 и 10 байт (в версии 4.0 эти данные передаются через стек сопроцессора 8087/80287, начиная с версии 5.0 — через стек центрального процессора 8086/80486). Во всех остальных случаях, если длина внутреннего представления больше 4 байт, соответствующий параметр передается по ссылке.

Ассемблерные функции в зависимости от длины внутреннего представления результата должны возвращать его через регистры центрального процессора или сопроцессора по следующим правилам:

- длиной в 1 байт — в регистре *AL*;
- длиной в 2 байта — в регистре *AX*;
- длиной в 4 байта — в регистрах *DX:AX* (старшее слово в *DX*);
- тип *REAL* (6 байт) — в регистрах *DX:BX:AX*;
- типы *SINGLE*, *DOUBLE*, *EXTENDED* и *COMP* — через стек сопроцессора 8087/80486;
- указатели — в регистрах *DX:AX* (сегмент в *DX*);
- строки возвращаются по ссылке: адрес начала строки помещается в *DX:AX* (сегмент в *DX*).

Все ассемблерные процедуры должны размещаться в сегменте с именем *CODE* или *CSEG*, или с именем, оканчивающимся на *_TEXT*; инициализированные локальные переменные помещаются в сегмент с именем *CONST* или с именем, оканчивающимся на *_DATA*. Все другие локальные переменные необходимо размещать в сегменте с именем *DATA* или *DSEG*, или с именем, оканчивающимся на *_BSS*. Любые другие объявления сегментов игнорируются. Все имена, объявленные в интерфейсной части модулей программы, написанной на Турбо Паскале, становятся доступны ассемблерной процедуре (функции) после их объявления директивой *EXTRN*. Точно так же все имена ассемблерных процедур и функций, которые должны быть доступны программе на Турбо Паскале, следует объявлять директивой *PUBLIC*.

11.2. ИСПОЛЬЗОВАНИЕ ВСТРОЕННЫХ МАШИННЫХ КОДОВ

В Турбо Паскале имеется возможность непосредственного включения в программу небольших фрагментов, написанных в машинных кодах. Для этого используется стандартная директива *INLINE*, за которой в круглых скобках следует один или несколько элементов машинного кода, разделенных косыми чертами. Элемент кода, в свою очередь, строится из одного или более элементов данных, разделенных знаками «+» или «-».

В качестве элемента данных может использоваться целая константа, идентификатор (переменной, константы или функции) или ссылка на счетчик адреса («*»). Каждый элемент данных вызывает генерацию 1 или 2 байт кода программы. Значение этого кода получается сложением или вычитанием элементов данных в соответствии с разделяющим их знаком. Значением идентификатора переменной, константы, функции служит адрес соответствующего объекта, значением ссылки на счетчик адреса является тот адрес, по которому будет размещаться следующий байт кода.

Элемент кода будет генерировать 1 байт кода, если этот элемент состоит только из целых констант и значение результата не превышает мощности одного байта, т.е. находится в диапазоне от 0 до 255. Если значение превышает 255 или элемент кода содержит ссылку на счетчик адреса, генерируются 2 байта. Знаки «<» и «>» могут использоваться для отмены автоматического выбора размера генерируемого кода. Если элемент кода начинается со знака «<», в код заносится только 1 байт (с младшей значимостью), даже если само значение занимает 2 байта. Наоборот, если элемент начинается со знака «>», в код заносятся 2 байта (старший байт может оказаться нулевым).

Значением идентификатора является смещение соответствующего объекта. Если переменная — глобальная, смещение задается относительно сегмента данных, хранящегося в регистре DS, если это локальная переменная, — относительно сегмента стека (регистр SP). Базовым сегментом типизированной константы является сегмент кода (регистр CS).

В следующем примере приводятся две короткие процедуры, с помощью которых можно ввести или вывести данные через любой порт ПК.

```
Function InPort (Port: Word): Word;
var
  pp: Word;
  cc: Char;
begin
  pp:=port;
  inline (
    $8b/$96/pp/           { mov  DX,pp[bp] }
    $EC/                  { IN   AX,DX }
    $88/$86/cc);          { mov  cc[bp],AX }
    InPort:=ord(cc);
end;
Procedure OutPort (Port, Bt: Word);
```

```

var
  pp: Word;
  cc: Char;
begin
  pp:=port;
  cc:=chr(Bt);
  inline (
    $8a/$86/cc/           { mov  AX,cc[bp] }
    $8b/$96/pp/           { mov  DX,pp[bp] }
    $EE)                  { OUT  DX,AX }
end;

```

Операторы **INLINE** могут произвольным образом смешиваться с другими операторами Турбо Паскаля, однако при выходе из процедуры (функции) содержимое регистров BP, SP, DS и SS должно быть таким же, как и при входе в нее.

С помощью директивы **INLINE** можно также задавать последовательность машинных кодов, которую необходимо несколько раз вставить в программу. Для этого используется описание **INLINE**—процедуры, на пример:

```

Procedure DisableInterrupts;
  inline ($FA); {CLI}

```

INLINE—процедура имеет обычный для Турбо Паскаля заголовок, в то время как тело процедуры пишется целиком с помощью оператора **INLINE**. Всякий раз, когда в программе будет встречаться оператор вызова **INLINE**—процедуры, компилятор Турбо Паскаля будет вставлять на это место не код вызова процедуры, а нужные машинные коды. Например, вместо вызова процедуры в операторе

```
DisableInterrupt;
```

компилятор вставит команду запрета прерываний **CLI**. Таким образом, **INLINE**—процедуры служат своеобразным средством расширения возможностей стандартного компилятора Турбо Паскаля и подобны макросам ассемблера. Использование **INLINE**—процедур увеличивает скорость исполнения программы, так как не осуществляется генерация (и исполнение) команд передачи управления в процедуру. По этой причине в **INLINE**—процедурах не следует использовать команды выхода из подпрограммы. **INLINE**—процедура может иметь параметры, однако на них нельзя ссылаться в **INLINE**—директивах (на другие символы Турбо Паскаля ссылаться можно). В следующем примере перемножаются два числа типа **INTEGER**, результат имеет тип **LONGINT**:

```

FUNCTION LongMul(X,Y : Integer) : Longint;
  inline (
    $5A/           { POP  AX; получить в AX число X }
    $58/           { POP  DX; получить в DX число Y }
    $F7/$EA);      { IMUL DX; DX:AX := X * Y }

```

Отметим, что в силу упоминавшегося сходства с макросами ассемблера, имена **INLINE** — подпрограмм не могут использоваться в качестве аргументов в операторах **@** или служить параметрами функций **ADDR**, **OFS** и **SEG**.

11.3. ОБРАЩЕНИЕ К ФУНКЦИЯМ ОПЕРАЦИОННОЙ СИСТЕМЫ

Турбо Паскаль предоставляет программисту практически неограниченные возможности использования любых функций стандартной операционной системы **MS-DOS**. При внимательном анализе материала этой книги Вы, очевидно, заметите, что значительную его часть составляет описание многочисленных библиотечных процедур и функций. Собственно язык Паскаль весьма прост и лаконичен, что, по мнению многих специалистов, и послужило одной из причин его широкого распространения. Библиотечные же процедуры и функции, в своей значительной части, являются, по существу, своеобразным интерфейсом между языковыми средствами Турбо Паскаля и функциями операционной системы. Разумеется, можно только приветствовать усилия разработчиков Турбо Паскаля по созданию мощных библиотек **TURBO.TPL** и **GRAPH.TPU**, однако ясно, что таким способом невозможно запрограммировать все допустимые обращения к средствам ДОС. Вот почему в Турбо Паскаль включены две процедуры, с помощью которых программист может сам сформировать вызов той или иной функции дисковой операционной системы (ДОС).

Следует учесть, что единственным механизмом обращения к функциям ДОС является инициация программного прерывания. *Прерывание* — это особое состояние вычислительного процесса. В момент прерывания нарушается нормальный порядок выполнения команд программы и управление передается специальной процедуре, которая входит в состав ДОС и называется процедурой обработки прерывания. Каждое прерывание характеризуется в рамках ДОС порядковым номером и связано со своей процедурой обработки. В архитектуре центрального процессора ПК предусмотрены прерывания двух типов — аппаратные и программные. Аппаратные прерывания создаются схемами контроля и управления ПК и сигнализируют операционной системе о переходе какого-либо устройства в новое состояние или о возникновении неисправности. Программные прерывания инициируются при выполнении одной из двух специальных команд микропроцессора (**INT** или **INTO**) и служат для обращения к средствам ДОС.

Описываемые ниже процедуры входят в состав библиотечного модуля **DOS.TPU** и становятся доступными после объявления **USES DOS**. При возникновении программного прерывания в большинстве случаев необходимо передать процедуре обработки прерывания некоторые параметры, в которых конкретизируется запрос нужной функции. Эти параметры, а также выходная информация (результат обработки прерывания) передаются от программы к процедуре и обратно через регистры центрального процессора. В составе модуля **DOS.TPU** для этих целей определен специальный тип:


```

type
  Registers = record
    case integer of
      0 : (AX, BX, CX, BP, SI, DI, DS, ES, Flags : word);
      1 : (AL, AH, BL, BH, CL, CH, DL, DH : byte)
    end;

```

Этот тип имитирует регистры центрального процессора и дает возможность обращаться к ним как к 16-битным или 8-битным регистрам.

Процедура INTR. С помощью этой процедуры иницируется программное прерывание с требуемым номером. Обращение:

INTR (<N>, <регистры>)

Здесь <N> — выражение типа *BYTE*; номер прерывания;

<регистры> — переменная типа *REGISTERS*; в этой переменной процедуре обработки прерывания передается содержимое регистров и в ней же возвращается выходная информация.

Например, прерывание с номером 18 (\$12) возвращает в регистре AX объем оперативной памяти ПК. Короткая программа, представленная в примере 11.1, выведет на экран сообщение об этом объеме.

Пример 11.1.

```

Uses DOS;
var
  r : registers;
begin
  Intr ($12, r);
  writeln ('Объем памяти = ', r.AX, ' Кбайт')
end.

```

Процедура MSDOS. Иницирует прерывание с номером 33 (\$21). Формат обращения:

MSDOS (<регистры>)

Программное прерывание с номером 33 (\$21) стоит особняком, так как оно дает доступ к большому количеству функций ДОС (этим прерыванием вызывается 85 функций). Рассматриваемая процедура полностью эквивалентна вызову процедуры *INTR* с номером прерывания 33. Например, следующая программа (пример 11.2) выведет на экран версию операционной системы:

Пример 11.2

```

Uses DOS;
var
  r : registers;
begin
  r.AH := $30;

```

```
MsDos(r);  
WriteLn('Версия операционной системы: ',  
        r.AL, '.' r.AH)  
end.
```

11.4. ПОДДЕРЖКА ПРОЦЕДУР ОБРАБОТКИ ПРЕРЫВАНИЙ

При написании процедур обработки прерываний существенными являются два обстоятельства. Во-первых, процедура обработки прерывания не должна искажать работу прерванной программы. Для этого необходимо сначала сохранить регистры центрального процессора, а перед выходом из процедуры — восстановить их. Во-вторых, процедура должна строиться по принципу реентерабельности (повторной входимости): ее работа может быть прервана в любой момент другими прерываниями и ДОС может обратиться к соответствующей функции до завершения обработки предыдущего прерывания.

Турбо Паскаль предоставляет программисту возможность написания процедур обработки прерывания на языке высокого уровня, хотя обычно такие процедуры пишутся на языке ассемблера.

Процедура обработки прерывания, написанная на Турбо Паскале, должна начинаться стандартной директивой **INTERRUPT** (прерывание), например:

```
Procedure IntProc (Flags, CS, IP, AX, BX, CX, DX,  
                  SI, DF, DS, ES, BP : word); interrupt;  
begin  
  ...  
end;
```

Формальные параметры в заголовке процедуры должны перечисляться в указанном порядке — через эти параметры все регистры прерванной программы становятся доступны процедуре обработки прерывания. Количество перечисляемых в заголовке процедуры параметров — регистров может быть любым, но не больше 12. Если в списке опущен какой-либо параметр, должны быть опущены также и все предшествующие ему параметры. Например, описание

```
Procedure IntProc(SI, DP, ES: word); interrupt;
```

будет неверным (опущены параметры DS и BP); правильное описание:

```
Procedure IntProc(SI, DP, DS, ES, BP: word); interrupt;
```

Заметим, что компилятор не контролирует порядок перечисления параметров в заголовке процедуры обработки прерывания.

Директива **INTERRUPT** вызывает генерацию специальных машинных кодов, обеспечивающих заталкивание регистров в стек при входе в процедуру и извлечение их из стека перед выходом из нее.

При входе в процедуру:

```
push ax  
push bx
```

```

push  cx
push  dx
push  si
push  di
push  ds
push  es
push  bp
mov   bp, si
sub   sp, LocalSize
mov   ax, SEG DATA
mov   ds, ax

```

При выходе из процедуры:

```

mov sp, bp
pop bp
pop es
pop ds
pop di
pop si
pop dx
pop cx
pop bx
pop ax
irep

```

В самой процедуре обработки прерывания не рекомендуется обращаться к другим функциям ДОС, так как некоторые из них, в том числе все функции ввода — вывода, нереентерабельны.

Для связи с любыми процедурами прерываний, а следовательно, и с процедурами, написанными программистом, используются векторы прерываний — четырехбайтные абсолютные адреса точек входа в эти процедуры. Векторы прерываний располагаются в младших адресах оперативной памяти, начиная с нулевого адреса: прерывание номер 0 — по адресу 0, номер 1 — по адресу $1 \cdot 4 = 4$, номер N — по адресу $N \cdot 4$. С помощью следующих двух процедур программист может прочитать содержимое любого вектора или установить его новое значение.

Процедура GETINTVEC. Возвращает вектор прерывания с указанным номером. Обращение:

```
GETINTVEC (<, <вектор>)
```

Здесь <D> — выражение типа *BYTE*; номер прерывания;

<вектор> — переменная типа *POINTER*; адрес точки входа в процедуру обработки прерывания.

Представленная в примере 11.3 программа выводит на экран содержимое всех ненулевых векторов прерываний.

Пример 11.3

Uses DOS;

```
var
  i : byte;
  p : pointer;
begin
  for i := 0 to 255 do
    begin
      GetIntVec (i, p);
      if (Seg (p^) <> 0) or (Ofs (p^) <> 0) then
        WriteLn (' N =', i:3, ' Seg =', Seg (p^):5,
                  ' Ofs =', Ofs (p^):5)
    end
  end.
```

Процедура SETINTVEC. Устанавливает новое значение вектора прерывания. Формат обращения:

SETINTVEC (<, <адрес>)

Здесь <D> — выражение типа *BYTE*; номер прерывания;

<адрес> — выражение типа *POINTER*; адрес точки входа в процедуру обработки прерывания.

При нормальном завершении программы она выгружается из памяти, что делает невозможным разработку резидентных в памяти процедур обработки прерываний. Вы можете прекратить работу программы и оставить ее резидентной в памяти, если воспользуетесь процедурой *KEEP*.

Процедура KEEP. Завершает работу программы и оставляет ее резидентной в памяти. Обращение:

KEEP (<код>)

Здесь <код> — выражение типа *WORD* — код завершения программы. Код завершения представляет собой фактически единственный механизм передачи сообщений от запущенной программы к программе, которая ее запустила. Он может быть проанализирован в вызывающей программе с помощью функции *DOSEXITCODE*.

Функция DOSEXITCODE. Возвращает значение типа *WORD* — код завершения подчиненной программы. Обращение:

DOSEXITCODE

11.5. ЗАПУСК ВНЕШНИХ ПРОГРАММ

Из программы, написанной на Турбо Паскале, можно запустить любую другую готовую к работе программу. Для этого используется процедура *EXEC* из библиотечного модуля *DOS*. Формат обращения к процедуре:

EXEC (<имя>, <параметры>)

Здесь *<имя>* — выражение типа *STRING*; имя файла с вызываемой программой;

<параметры> — выражение типа *STRING*; параметры вызова.

Имени запускаемой программы может предшествовать путь к файлу. Параметры передаются запускаемой программе в виде текстовой строки и могут быть проанализированы ею с помощью двух следующих функций.

Функция *PARAMCOUNT*. Возвращает общее количество параметров вызова программы (значение типа *WORD*). Обращение:

PARAMCOUNT

Параметры вызова обычно следуют в командной строке ДОС сразу за именем вызываемой программы и отделяются от этого имени и друг от друга пробелами, например:

```
C:\TP\TURBO MYPROG.PAS
C:\SIAM A:\SYSTEM1.SIA
```

Здесь *MYPROG.PAS* и *A:\SYSTEM1.SIA* — параметры, передаваемые программам *TURBO* и *SIAM*.

При вызове программы непосредственно из среды Турбо Паскаля ей можно передать параметры с помощью опции *OPTIONS/PARAMETERS* (см. прил.1).

Функция *PARAMSTR*. Возвращает значение типа *STRING*, соответствующее нужному параметру вызова. Формат обращения:

PARAMSTR (<D>)

Здесь *<D>* — выражение типа *WORD*; порядковый номер параметра.

Заметим, что программе всегда передается параметр, соответствующий *N = 0*. В этом параметре ДОС сообщает полное имя запущенной программы с указанием диска и каталога, откуда она была загружена.

Использование процедуры *EXEC* имеет ряд особенностей. Прежде всего необходимо отметить, что сама вызывающая программа остается резидентной в памяти, поэтому она не должна занимать всю оперативную память. Объем выделяемой программе памяти регулируется опцией *OPTIONS/MEMORY SIZES* (см. прил.1). По умолчанию параметры *LOW HEAP LIMIT* и *HIGH HEAP LIMIT* этой опции таковы (соответственно 0 и 655360 байт), что вызывающая программа, написанная на Турбо Паскале, занимает весь доступный объем памяти, и вызываемая программа не будет загружена. Полезно включить в текст вызывающей программы директиву компилятора, в которой изменяются принятые по умолчанию размеры памяти. Например, так:

{\$M 2048, 0, 0}

Такая директива ограничивает используемую программой область стека величиной 2 Кбайта и исключает возможность использования в ней динамической памяти. Разумеется, Вы можете установить и другие значения параметров в этой директиве.

Специфические особенности исполнения программ Турбо Паскаля требуют изменения стандартных значений некоторых векторов прерываний. К ним относятся векторы со следующими шестнадцатеричными номерами:

\$00, \$02, \$18, \$23, \$24, \$34, \$35, \$36, \$37,
\$38, \$39, \$3A, \$3B, \$3C, \$3D, \$3E, \$3F, \$75.

Начальные значения этих векторов сохраняются в восемнадцати переменных с именами *SAVEINTXX* из библиотечного модуля *SYSTEM*, где *XX* — шестнадцатеричный номер прерывания. Поэтому непосредственно перед запуском внешней программы и сразу после возврата из нее рекомендуется вызывать библиотечную процедуру без параметров *SWAPVECTORS*, которая обменивает содержимое векторов прерывания и перечисленных переменных.

Программа из примера 11.4 читает с клавиатуры любую команду ДОС, затем вызывает командный процессор *COMMAND.COM* операционной системы и передает ему эту команду.

Обратите внимание: для указания файла *COMMAND.COM* и пути к нему используется обращение к библиотечной функции *GETENV*, с помощью которой можно получить параметры настройки операционной системы. В частности, параметр *COMSPEC* определяет спецификацию файла, содержащего командный процессор.

Пример 11.4

```
{ $M 1024, 0, 0 }
Uses DOS;
var
  st : string [79];
begin
  write ('Введите команду ДОС: ');
  readln (st);
  if st <> '' then
    begin
      st := '/C '+st;
      SwapVectors;
      Exec (GetEnv ('COMSPEC'), st);
      SwapVectors;
    end
  end.
```

Функция ENVCOUNT. Возвращает значение типа *INTEGER*, в котором содержится общее количество установленных в ДОС параметров. Обращение:

ENVCOUNT

Функция ENVSTR. Возвращает значение типа *STRING*, содержащее имя и значение нужного параметра настройки операционной системы. Формат обращения:

ENVSTR (<D>)

Здесь <D> — выражение типа *INTEGER*; номер параметра.

Эта функция возвращает строку типа *NAME=VALUE*, где *NAME* — имя, а *VALUE* — значение соответствующего параметра настройки.

Функция GETENV. Возвращает значение типа *STRING*, в котором содержится параметр настройки ДОС. Формат обращения:

GETENV (<имя>)

Здесь <имя> — выражение типа *STRING*; имя параметра.

Эта функция имеет параметр обращения *NAME*, а возвращает значение *VALUE* (см. функцию *ENVSTR*).

11.6. ОВЕРЛЕЙ

Как отмечалось в гл.9, максимальный размер модуля не может превышать 64 Кбайта, однако количество модулей не ограничено, что дает возможность разрабатывать весьма крупные программы, занимающие, например, всю доступную оперативную память ПК (приблизительно 580 Кбайт). Тем не менее, в некоторых случаях и этот объем может оказаться недостаточным. Турбо Паскаль предоставляет в распоряжение программиста простой и достаточно эффективный механизм оверлея, с помощью которого можно создавать программы практически неограниченной длины (следует оговориться, что речь идет только о длине кода программы; два важных размера — длина сегмента данных и размер программного стека — в Турбо Паскале не могут превышать 64 Кбайта независимо от структуры программы).

Оверлей — это такой способ использования оперативной памяти, при котором в один и тот же участок памяти, называемый оверлейным буфером, попеременно по мере надобности загружаются различные оверлейные (перекрывающиеся) модули. При этом все оверлейные модули в готовом к работе виде хранятся на диске, а в оперативной памяти в каждый момент находится лишь один активный модуль и, возможно, небольшое число неактивных.

Пусть, например, программа (рис. 11.1) состоит из главной части *MAIN* и двух модулей *A* и *B*, а *LM*, *LA* и *LB* — соответственно длина главной части и обоих модулей, причем $LA > LB$. Тогда неоверлейная программа займет в памяти $LM + LA + LB$ байт, в то время как оверлейная программа — лишь $LM + LA$ байт.

При исполнении оверлейной программы в память первоначально загружается главная часть и один из модулей, например, модуль *A*. Если в процессе исполнения программы встретится обращение к модулю *B*, программа приостановит свою работу, с диска в оверлейный буфер будет загружен модуль *B* (модуль *A* при этом частично уничтожается), после чего программа продолжит свою работу. Если в дальнейшем встретится обращение к *A*, точно таким же образом будет загружен модуль *A*, причем загрузка нужных модулей в оверлейный буфер осуществляется автоматически и программисту не нужно об этом заботиться.

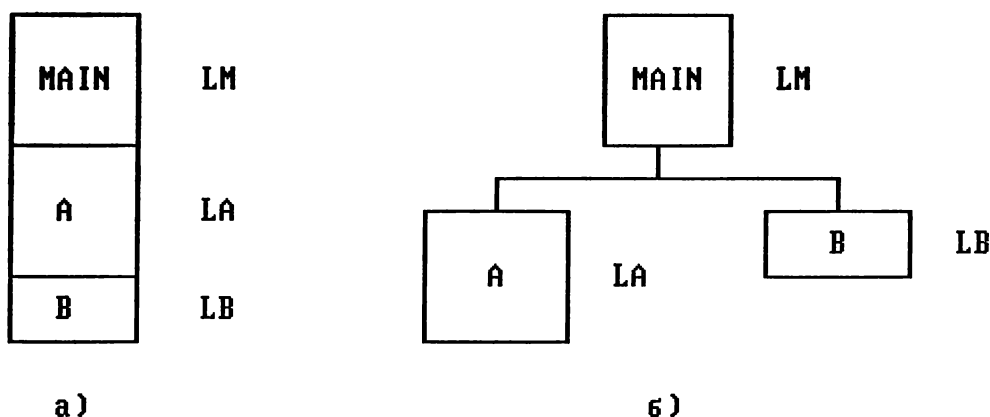


Рис.11.1. Пример структуры программы:
 а) неоверлейная; б) оверлейная

Описанный механизм выявляет главное преимущество оверлейной структуры: объем оперативной памяти, занимаемой оверлейной программой, определяется длиной ее главной части и наибольшего из перекрывающихся модулей, в то время как при неоверлейной структуре в этот объем входит суммарная длина всех модулей. Чем больше в программе оверлейных модулей и чем меньше длина наибольшего из них, тем больший выигрыш в памяти дает оверлейная структура. Однако совершенно очевиден и главный недостаток таких структур: на каждую загрузку оверлейного модуля с диска в оверлейный буфер требуется дополнительное время, поэтому оверлейная программа будет исполняться с меньшей скоростью.

Работа оверлейных программ обеспечивается с помощью процедур и функций библиотечного модуля *OVERLAY*, входящего в библиотечный файл *TURBO.TPL*.

При создании оверлейных программ нужно руководствоваться следующей последовательностью действий.

- Выделить главную часть программы и разбить оставшуюся часть на несколько модулей. Отметим, что никаких дополнительных ограничений на модули, по сравнению с описанными в гл.9, не накладывается за одним исключением: в оверлейных модулях нельзя использовать процедуры обработки прерываний. Желательно продумать состав модулей таким образом, чтобы по возможности минимизировать количество их перезагрузок в оверлейный буфер в процессе исполнения программы.
- В главной части программы указать с помощью директив компилятора вида `{ $O <имя> }` те модули, которые будут оверлейными, например.

```
Program Main;
Uses CRT, DOS, Graph, Overlay, UnitA, UnitB;
{ $O DOS }
{ $O UnitA }
```


{*\$O UnitB*}

Следует подчеркнуть, что из всех стандартных библиотечных модулей только один модуль *DOS* может быть оверлейным, остальные модули (*CRT*, *Graph*, *Printer* и т.д.) не могут объявляться оверлейными.

- Предусмотреть перед первым по логике работы программы обращением к оверлейному модулю вызов процедуры инициализации оверлея *OVRINIT*. Здесь же, если это необходимо, следует установить размер оверлейного буфера и указать возможность использования расширенной памяти (см. ниже).
- В начале главной программы и каждого оверлейного модуля необходимо поместить директивы компилятора *{*\$O+*}* и *{*\$F+*}* или установить опции *OPTIONS /COMPILE/FORCE FAR CALLS* и *OPTIONS/COMPILE/ OVERLAYS ALLOWED* (см. прил.1) в активное состояние, после чего откомпилировать программу на диск. Программа готова к работе.

Таким образом, все процедуры и функции в оверлейной программе должны использовать дальнюю модель вызова — это обязательное условие. Отметим, что попытка компиляции оверлейного модуля, в начале которого отсутствует директива *{*\$O+*}* (предполагается, что опция среды *OPTIONS/COMPILE/OVERLAY ALLOWED* неактивна), будет обнаружена компилятором, в то время как неправильная (ближняя) модель вызова оверлейных подпрограмм компилятором не контролируется и может привести к непредсказуемым результатам при исполнении программы.

Далее, инициация оверлея (осуществляется вызовом процедуры *OVRINIT*, см. ниже) должна происходить до вызова любого из оверлейных модулей. Это требование кажется тривиальным, однако множество проблем в оверлейных программах обычно связано именно с ним. Дело в том, что обращение к оверлейному модулю может происходить еще до начала работы основной программы: напомним, что любой модуль (в том числе и оверлейный) может иметь иницирующую часть, которая исполняется перед началом работы основной программы. В связи с этим рекомендую придерживаться следующего простого правила: никогда не используйте оператор *BEGIN* в конце модуля, если Вам нет нужды в иницирующих действиях; пустая иницирующая часть *содержит пустой оператор*, которому будет передано управление на этапе инициации. Таким образом, пустая иницирующая часть оверлейного модуля очень часто может вызывать сообщение об ошибке периода исполнения с кодом 208 (не установлена система управления оверлеем). Как же быть, если в оверлейном модуле все-таки нужна иницирующая часть? В этом случае можно рекомендовать следующий прием. Создайте лишний модуль, в котором будут пустыми все части, кроме иницирующей. В этой части разместите команды инициации оверлея. Новый модуль не должен быть оверлейным и его имя должно стоять в предложении *USES* основной программы перед именем любого оверлейного модуля. После компиляции такой программы инициация оверлея будет осуществляться перед выполнением любой другой иницирующей части и проблема будет решена.

Процедура `OVRINIT`. Инициализирует оверлейный файл. Обращение:

`OVRINIT (<имя>)`

Здесь `<имя>` — выражение типа **STRING**; имя файла с оверлейной частью программы.

При компиляции оверлейной программы создается специальный файл с именем, совпадающим с именем главной программы, и расширением `.OVR`. В этот файл компилятор помещает все оверлейные модули, из него же эти модули загружаются в оверлейный буфер в процессе исполнения программы. Файл с оверлейной частью программы должен размещаться в том же каталоге, что и файл с главной частью (с расширением `.EXE`). Отметим, что имя оверлейного файла необходимо дополнять расширением `.OVR`.

Обычно размер оверлейного буфера определяется автоматически таким образом, чтобы в нем мог разместиться самый крупный из всех оверлейных модулей. Программист может увеличить размер буфера. Тогда при загрузке в буфер очередного модуля программа проверит, достаточно ли в буфере свободного места и, если достаточно, загрузит новый модуль сразу за старым, который, таким образом, не будет уничтожен. Такой механизм способствует минимизации потерь времени на перезагрузку модулей. Если установлен очень большой размер буфера, то в нем, возможно, смогут разместиться все оверлейные модули, однако в этом случае оверлейная структура становится просто ненужной.

Процедура `OVRSETBUF`. Устанавливает размер оверлейного буфера. Формат обращения:

`OVRSETBUF (<длина>)`

Здесь `<длина>` — выражение типа **LONGINT** задает новую длину буфера в байтах не больше той, которую устанавливает сама система автоматически. Расширение буфера идет за счет соответствующего уменьшения доступной динамической памяти, поэтому к моменту вызова этой процедуры куча должна быть пустой.

Функция `OVRGETBUF`. Возвращает значение типа **LONGINT**, содержащее текущий размер оверлейного буфера. Обращение:

`OVRGETBUF`

Процедура `OVRINITEMS`. Обеспечивает использование расширенной памяти. Если Ваш ПК относится к классу компьютеров типа IBM PC/AT и в нем имеется так называемая **EMS**—память (*Expanded Memory Specification* — расширенная¹ память, удовлетворяющая стандарту фирм Lotus/Intel/Microsoft). Вы можете использовать эту память для размещения в ней оверлейного файла `.OVR`. Поскольку время доступа к расширенной памяти значительно меньше времени чтения с диска, такое размещение увеличивает скорость исполнения оверлейной программы. Обращение:

`OVRINITEMS`

¹ Слово *расширенная* — букальный перевод англ. *expanded*. В отечественной литературе для **EMS**—памяти часто используется термин *отображаемая*.

При обращении к этой процедуре программа прежде всего проверит, достаточен ли объем имеющейся в Вашем ПК *EMS*—памяти для размещения оверлейного файла. Если это так, то оверлейный файл считывается в *EMS*—память, сам файл закрывается, и программа будет получать оверлейные модули из этой памяти. Если же *EMS*—память отсутствует или ее объем недостаточен, обращение к процедуре игнорируется, и программа будет считывать оверлейные модули с диска.

Все управление оверлеем осуществляется стандартной подпрограммой, которая называется *администратором оверлея*. Эта подпрограмма получает управление всякий раз, когда программа обращается к ресурсам оверлейного модуля, не размещенного в данный момент в буфере. Администратор оверлея сначала перемещает предыдущий оверлейный модуль из буфера в так называемую контрольную зону, а уже затем грузит с диска в буфер новый модуль. Если в момент, когда оверлей находится в контрольной зоне, программа вновь обратится к нему, он вернется на свое старое место и таким образом затраты времени на обмен данными с диском будут уменьшены. Программист может задать размер контрольной зоны с помощью обращения к процедуре *OVRSETRETRY* и получить этот размер с помощью функции *OVRGETRETRY*. Обычно размер контрольной зоны составляет от одной трети до половины размера оверлейного буфера. Вы можете подобрать этот размер экспериментально в ходе пробного прогона программы. Для этого используются две переменные модуля *OVERLAY*:

var

```
OvrTrapCount: Word; {Счетчик обращений к администратору}
OvrLoadCount: Word; {Счетчик загрузок в оверлейный буфер}
```

Всякое обращение программы к оверлейному модулю, которого нет в оверлейном буфере (в том числе и к модулю, находящемуся в контрольной зоне), приводит к наращиванию содержимого переменной *OVRTRAPCOUNT* на единицу. Всякая загрузка оверлейного модуля из файла в буфер увеличивает на единицу счетчик *OVRLOADCOUNT*.

В интерфейсной части модуля *OVERLAY* объявлены еще три переменные, которые могут оказаться полезными для некоторых применений.

type

```
vrReadFunc = Function (OvrSeg: Word): Integer;
```

var

```
vrReadBuf : OvrReadFunc;      {Функция чтения из
                                оверлейного файла}
vrResult   : Integer;          {Признак ошибки оверлея}
vrFileMode: Byte;              {Способ доступа к оверлейному файлу}
```

Переменная *OVRRRESULT* содержит код, указывающий на успех или неуспех каждой очередной операции администратора оверлея. Значения этой переменной могут быть такими:

- 0 — операция прошла успешно;
- 1 — общая ошибка;
- 2 — не найден оверлейный файл;

- 3 — не хватает памяти для оверлейного буфера;
- 4 — ошибка чтения — записи оверлейного файла;
- 5 — не работает драйвер *EMS* — памяти;
- 6 — не хватает *EMS* — памяти.

Переменная *OVRFILEMODE* обычно содержит 0, что трактуется как возможность доступа к оверлейному файлу только для чтения информации. Перед вызовом процедуры *OVRINIT* программа может установить другое значение этой переменной и таким образом изменить доступ к файлу, что бывает необходимым, если ПК подключен к сети ЭВМ.

В переменной *OVRREADBUF* содержится имя функции, к которой обращается администратор оверлея при каждом чтении из оверлейного файла. Программа может перехватить обращение к этой функции и проанализировать результат операции. Для этого необходимо в основной программе сохранить имя стандартной функции чтения в глобальной переменной типа *OVRREADFUNC* и поместить в переменную *OVRREADBUF* имя новой функции. В эту новую функцию администратор будет передавать управление всякий раз, когда появится необходимость чтения из оверлейного файла. Программа может проверить состояние дисков перед исполнением операции (например, наличие нужного сменного диска), выполнить саму операцию (путем вызова функции, сохраненной в глобальной переменной), проверить результат обращения и предпринять необходимые действия. Отметим, что нормальное завершение операции чтения указывается нулевым значением функции чтения, ненулевое значение означает ту или иную ошибку; код ошибки стандартен для ДОС (см. прил.3) и/или для драйвера *EMS* — памяти.

При желании Вы можете пристыковать оверлейный файл в конец *EXE*-файла основной программы. При этом следует учесть, что интегрированная среда пристыковывает в конец *EXE*-файла отладочные таблицы, поэтому программу и все ее модули следует компилировать в режиме отключенных опций *OPTIONS /COMPILER/DEBUG INFORMATION* и *OPTIONS/COMPILER/LOCAL SYMBOLS* (см.прил.1.2.8). Для объединения *EXE*-файла с оверлейным файлом необходимо дать такую команду ДОС:

```
COPY /B NAME.EXE+NAME.OVR
```

Здесь *NAME.EXE* — имя *EXE*-файла, *NAME.OVR* — имя оверлейного файла. Чтобы оверлеи читались из *EXE*-файла, нужно просто указать имя этого файла при обращении к *OVRINIT*:

```
OvrInit (ParamStr(0));
```

(в программу всегда передается параметр *ParamStr(0)*, в котором ДОС сообщает полное имя запущенной программы — с указанием диска и каталога, откуда была загружена программа).

11.7. ПРЯМОЕ ОБРАЩЕНИЕ К ПАМЯТИ И ПОРТАМ ВВОДА-ВЫВОДА

В Турбо Паскале имеется пять предварительно объявленных массивов: *MEM*, *MEMW*, *MEML*, *PORT* и *PORTW*. Первые три обеспечивают доступ к

любому участку оперативной памяти по абсолютному адресу, два других — доступ к портам ввода — вывода.

Компонентами массива *MEM* являются данные типа *BYTE*, массива *MEMW* — типа *WORD*, массива *MEML* — типа *LONGINT*. Обращение к элементам этих массивов, т.е. их индексация, имеет специальный вид: каждый индекс представляет собой абсолютный адрес и состоит из двух выражений типа *WORD*; первое дает сегментную часть адреса, второе — смещение; выражения разделяются двоеточием. Например:

```
Mem[$0000:$1000] := 0;
DataMem := MemW[Seg(p):Ofs(p)];
MemLong := MemL[64:i*SizeOf(real)];
```

Как следует из технического описания операционной системы *MS-DOS*, в памяти по адресу *\$F000:\$FFFE* располагается байт-указатель типа компьютера. Следующая программа (пример 11.5) прочтет этот байт и выведет на экран тип Вашего ПК.

Пример 11.5

```
begin
  Write (' Тип компьютера: ');
  case Mem [$F000:$FFFE] of
    $FF : WriteLn ('PC');
    $FE : WriteLn ('XT');
    $FD : WriteLn ('PCjr');
    $FC : WriteLn ('AT');
    $F9 : WriteLn ('совместимый с PC')
  end
end.
```

Компонентами массива *PORT* являются байты (тип *BYTE*), а массива *PORTW* — слова (тип *WORD*). Индексами этих массивов должно быть выражение типа *BYTE*, указывающее номер нужного порта. Присвоение значения элементу массива *PORT* или *PORTW* приведет к записи в порт, упоминание элемента в выражении — к чтению из порта. Компоненты массивов *PORT* и *PORTW* нельзя передавать в качестве параметров процедурам или функциям. Эти идентификаторы не употребляются без индексных выражений.

11.8. ДЛИННЫЕ СТРОКИ

Среда Турбо Паскаль 7.0 обладает весьма интересным новшеством: в ней введена поддержка длинных строк, т.е. строк, длина которых может превышать 255 байт. Как известно, тип *String* в Турбо Паскале имеет максимальную длину 255 байт. Это связано с тем, что истинная длина строки в этом типе указывается первым байтом, мощность которого не позволяет определять строки большей длины. В то же время в языке *C* используется другой подход к заданию текстовых строк: первый байт строки является ее первым символом, второй байт — вторым символом и т.д. пока не встретится байт с символом #0. Таким образом, длина строки

не указывается явно, как в типе *String*, а определяется по замыкающему символу #0. Ясно, что такие строки могут иметь произвольную длину, ограничиваемую лишь объемом оперативной памяти или принятой схемой ее адресации: в MS-DOS такой границей является длина сегмента, поэтому максимально возможная длина C-строки для ПК составляет 65535 символов. Такие строки в дальнейшем будем называть ASCIIZ-строками.

Для реализации операций над ASCIIZ-строками в язык введен новый тип *PChar*, определяемый как указатель на символ:

```
type
  PChar = ^Char;
```

Однако такой обычный для Паскаля тип-указатель в рамках Турбо Паскаля 7.0 трактуется необычным способом: считается, что он указывает на цепочку символов, заканчивающуюся терминальным нулем, т.е. на ASCIIZ-строку. Более того, с этим типом совместим любой одномерный символьный массив с нулевой левой границей, а переменные типа *PChar* можно индексировать, как если бы они были массивами символов. Следующая программа напечатает все заглавные буквы английского алфавита:

```
{ $X+ }           { Включаем расширенный синтаксис }
var
  Chars : array [0..26] of Char;      { Массив символов }
  PChars: PChar;                     { Указатель на символ }
  k      : Integer;
begin
  for k := 0 to 25 do
    Chars[k] := Chr(k+ord('A'));      { Наполняем массив }
    PChars := Chars;                  { Указателю присваиваем массив! }
    PChars[26] := #0;                 { Индексируем указатель! }
    WriteLn(PChars)                   { Печатаем указатель! }
end.
```

Три последние оператора программы недопустимы в стандартном Паскале и в ранних версиях Турбо Паскаля, но возможны в версии 7.0, если включен расширенный синтаксис (директивой *{ \$X+ }* или опцией *Options/Compiler/Extended syntax*). Обратите внимание: процедура *WriteLn* этой версии умеет работать с ASCIIZ-строками.

Для поддержки ASCIIZ-строк разработан модуль *Strings*, в котором реализованы необходимые процедуры и функции.

Функция StrCat. Объединяет строки. Заголовок:

```
Function StrCat(Dest, Source: PChar): PChar;
```

Копирует строку *Source* в конец строки *Dest* и возвращает указатель на начало *Dest*.

Функция StrComp. Сравнивает строки. Заголовок:

```
Function StrComp(Str1, Str2: PChar): Integer;
```

Побайтно сравнивает строку *Str1* со строкой *Str2* и возвращает следующий результат:

```

=0      Str1=Str2;
>0      Str1>Str2;
<0      Str1<Str2.

```

Функция *StrCopy*. Копирует строку. Заголовок:

```
Function StrCopy(Dest, Source: PChar): PChar;
```

Копирует строку *Source* в строку *Dest* и возвращает указатель на *Dest*. *StrCopy* не проверяет реальный размер памяти, связанный с *Dest* (он должен быть не меньше *StrLen(Source)+1*).

Процедура *StrDispose*. Удаляет строку из кучи. Заголовок:

```
Procedure StrDispose(Str: PChar);
```

Строка *Str* должна быть предварительно помещена в кучу функцией *StrNew*. Если *Str=NIL*, процедура ничего не делает.

Функция *StrECopy*. Объединяет строки. Заголовок:

```
Function StrECopy(Dest, Source: PChar): PChar;
```

Эта функция работает в точности как *StrCat*, но возвращает указатель на конец сцепленных строк, т.е. на терминальный ноль.

Функция *StrEnd*. Возвращает конец строки. Заголовок:

```
Function StrEnd(Str: PChar): PChar;
```

Функция возвращает указатель на терминальный ноль *ASCIIZ*—строки *Str*.

Функция *StrIComp*. Сравнивает строки. Заголовок:

```
Function StrIComp(Str1, Str2: PChar): PChar;
```

Функция сравнивает строки, игнорируя возможную разницу в высоте букв. Возвращает такой же результат, как и *StrComp*. Замечу, что функция правильно работает лишь с латиницей. Для кириллицы ее нужно модифицировать (см. ниже).

Функция *StrLCat*. Объединяет строки. Заголовок:

```
Function StrLCat(Dest, Source: PChar; MaxLen: Word): PChar;
```

Копирует символы строки *Source* в конец строки *Dest* до тех пор, пока не будут скопированы все символы или когда длина сцепленной строки *Dest* не достигнет *MaxLen*. Возвращает указатель на сцепленную строку.

В отличие от *StrCopy* эта функция блокирует возможное переполнение области памяти, связанной с *Dest*. Обычно в качестве *MaxLen* используется выражение *SizeOf(Dest)-1*. Например:

```

{$X+}
Uses Strings;
```

```
var
  S: array [0..9] of Char;
begin
  StrCopy(S, 'Turbo');
  StrLCat(S, ' ', SizeOf(S)-1);
  StrLCat(S, 'Pascal', SizeOf(S)-1);
  WriteLn(S)           {Напечатает "Turbo Pas"}
end;
```

Функция StrLComp. Сравнивает строки. Заголовок:

Function StrLComp(Dest, Source: PChar; MaxLen: Word): PChar;

В отличие от *StrComp* сравнивает не более *MaxLen* символов строк. Возвращаемый результат такой же, как и у *StrComp*.

Функция StrLCopy. Заголовок:

Function StrLCopy(Dest, Source: PChar; MaxLen: Word): PChar;

Копирует символы из строки *Source* в строку *Dest* до тех пор, пока не будет скопирована вся строка или пока не будет скопировано *MaxLen* символов. В отличие от *StrCopy* блокирует возможное переполнение области памяти, связанной с *Dest*. В качестве *MaxLen* обычно используется выражение *SizeOf(Dest)-1*. Например:

```
{ $X+ }
Uses Strings;
var
  S: array [0..9] of Char;
begin
  StrLCopy(S, 'Turbo Pascal', SizeOf(S)-1);
  WriteLn(S)           {Напечатает "Turbo Pas"}
end;
```

Функция StrLen. Возвращает длину строки. Заголовок:

Function StrCat(Str: PChar): Word;

Функция StrLIComp. Сравнивает строки с учетом регистра. Заголовок:

Function StrLIComp(Str1, Str2: PChar; MaxLen: Word): PChar;

Сравнивает не более *MaxLen* символов строк, проверяя точное соответствие высоты букв. Возвращаемый результат см. *StrComp*. Функция правильно работает только с латиницей.

Функция StrLower. Преобразует в строчные буквы. Заголовок:

Function StrLower(Str: PChar): PChar;

Преобразует заглавные буквы строки *Str* к строчным и возвращает указатель на результат. Функция правильно работает только с латиницей.

Функция StrMove. Копирует строку. Заголовок:

Function StrMove(Dest, Source: PChar; Count: Word): PChar;

Копирует точно *Count* символов строки *Source* в строку *Dest* и возвращает указатель на результат. Функция игнорирует действительные размеры строк и может выйти за их пределы.

Функция StrNew. Помещает строку в кучу. Заголовок:

Function StrNew(Str: PChar): PChar;

Функция StrPas. Преобразует ASCIIZ-строку в строку *String*. Заголовок:

Function StrPas(Str: PChar): String;

Функция StrPCopy. Преобразует строку *String* в ASCIIZ-строку. Заголовок:

Function StrPCopy(Str: PChar; S: String): PChar;

Возвращает указатель на *Str*.

Функция StrPos. Ищет подстроку. Заголовок:

Function StrPos(Str1, Str2: PChar): PChar;

Ищет подстроку *Str2* в строке *Str1* и возвращает указатель на первое вхождение *Str2* или *NIL*, если подстрока не найдена.

Функция StrRScan. Ищет последний символ. Заголовок:

Function StrRScan(Str: PChar; Ch: Char): PChar;

Ищет символ *Ch* в строке *Str* и возвращает указатель на последний обнаруженный символ *Ch* или *NIL*, если символ не найден.

Функция StrScan. Ищет первый символ. Заголовок:

Function StrScan(Str: PChar; Ch: Char): PChar;

Ищет символ *Ch* в строке *Str* и возвращает указатель на первый обнаруженный символ *Ch* или *NIL*, если символ не найден.

Функция StrUpper. Заголовок:

Function StrLower(Str: PChar): PChar;

Преобразует строчные буквы строки *Str* к заглавным и возвращает указатель на результат. Функция правильно работает только с латиницей.

Четыре функции модуля *Strings* (*StrLower*, *StrUpper*, *StrIComp* и *StrLIComp*) используют преобразование высоты букв и работают корректно только для букв латинского алфавита (латиницы). Для русских букв эти функции можно изменить следующим образом:

!Этот модуль содержит модификацию функций стандартного модуля *Strings* для работы с кириллицей (альтернативный

вариант кодировки}
 {\$X+}

Unit StringsR;

INTERFACE

Function LoCase(Ch: Char): Char;

Function UpCase(Ch: Char): Char;

Function StrLower(Str: PChar): PChar;

Function StrUpper(Str: PChar): PChar;

Function StrIComp(Str1, Str2: PChar): Integer;

Function StrLIComp(Str1, Str2: PChar; MaxLen: Word): Integer;

IMPLEMENTATION

Uses Strings;

Function LoCase(Ch: Char): Char;

{Преобразует латинскую или русскую букву Ch к строчной}

begin

case Ch **of**

'A'..'Z': LoCase := Chr(ord('a')+ord(Ch)-ord('A'));

'А'..'П': LoCase := Chr(ord('а')+ord(Ch)-ord('А'));

'P'..'Я': LoCase := Chr(ord('p')+ord(Ch)-ord('P'));

else

LoCase := Ch

end

end;

{-----}

Function UpCase(Ch: Char): Char;

{Преобразует латинскую или русскую букву Ch к заглавной}

begin

case Ch **of**

'a'..'z': UpCase := Chr(ord('A')+ord(Ch)-ord('a'));

'а'..'п': UpCase := Chr(ord('А')+ord(Ch)-ord('а'));

'p'..'я': UpCase := Chr(ord('P')+ord(Ch)-ord('p'));

else

UpCase := Ch

end

end;

{-----}

Function StrLower(Str: PChar): PChar;

{Преобразует латинские и русские буквы строки Str к строчным}

var

k: Word;

begin

for k := 0 **to** StrLen(Str)-1 **do**

Str[k] := LoCase(Str[k]);

StrLower := Str

end;

{-----}

```

Function StrUpper(Str: PChar): PChar;
{Преобразует латинские и русские буквы строки Str к заглавным}
var
    k: Word;
begin
    for k := 0 to StrLen(Str)-1 do
        Str[k] := UpCase(Str[k]);
    StrUpper := Str
end;
{-----}
Function StrIComp(Str1, Str2: PChar): Integer;
{Сравнивает две строки, игнорируя возможную разницу в высоте
латинских или русских букв}
var
    k: Word;
    Max: Word;
begin
    {Определяем максимальное количество сравниваемых символов
как минимум длин строк}
    Max := StrLen(Str1);
    if StrLen(Str2) < Max then
        Max := StrLen(Str2);
    {Проверяем символы до первого несовпадения.
Буквы преобразуем к заглавным}
    for k := 0 to Max-1 do
        if UpCase(Str1[k]) <> UpCase(Str2[k]) then
            begin
                {С.роки не равны}
                StrIComp := ord(UpCase(Str1[k])) - ord(UpCase(Str2[k]));
                Exit
            end;
    {Разницы нет - результат зависит от совпадения длин}
    StrIComp := StrLen(Str1) - StrLen(Str2)
end;
{-----}
Function StrLIComp(Str1, Str2: PChar; MaxLen: Word): Integer;
{Сравнивает не более MaxLen символов строк, проверяя точное
соответствие высоты букв}
var
    k: Word;
    Max: Word;
begin
    {Определяем максимальное количество сравниваемых символов
как минимум длин строк и MaxLen}
    Max := MaxLen;
    if StrLen(Str1) < Max then
        Max := StrLen(Str1);
    if StrLen(Str2) < Max then

```

```
Max := StrLen(Str2);  
{Проверяем символы до первого несовпадения.  
Буквы преобразуем к заглавным}  
for k := 0 to Max do  
  if UpCase(Str1[k]) <> UpCase(Str2[k]) then  
    begin  
      StrLComp := ord(UpCase(Str1[k])) -  
                  ord(UpCase(Str2[k]));  
      Exit  
    end;  
{Разницы нет. Если проверено MaxLen символов, строки  
считаются равными, в противном случае результат зависит  
от совпадения длин строк}  
if Max=MaxLen then  
  StrLComp := 0  
else  
  StrLComp := StrLen(Str1)-StrLen(Str2)  
end;  
end.
```

Если Вы будете использовать этот модуль, ссылайтесь на него в пред-
ложении **Uses** после ссылки на стандартный модуль *Strings*:

Uses Strings, StringsR, ...

ВСТРОЕННЫЙ АССЕМБЛЕР

Ассемблером называется машинно-зависимый компилятор, преобразующий специальным образом составленные текстовые строки в машинные инструкции. Как и любой другой компилятор, ассемблер упрощает разработку программ за счет того, что предоставляет пользователю доступ к кодам машинных инструкций и операндам с помощью символьных имен.

В этой главе рассматриваются приемы программирования с помощью ассемблера, встроенного в компилятор Турбо Паскаля. Встроенный ассемблер имеется в версиях 6.0 и 7.0 Турбо Паскаля и в руках опытного программиста представляет собой мощный инструмент, позволяющий «выжать» из ПК все возможное.

12.1. ОБЩЕЕ ОПИСАНИЕ МП 8086/8088

Встроенный ассемблер (далее — просто ассемблер) дает возможность программировать на уровне отдельных машинных инструкций. Это — главное отличие ассемблера от Паскаля и в этом отличии сосредоточены все его достоинства и недостатки. Достоинство заключается в том, что, программируя на ассемблере, программист обычно выбирает последовательность машинных инструкций так, чтобы реализовать нужные вычисления с максимальной скоростью при минимальных затратах памяти, в то время как даже такой весьма совершенный компилятор, как компилятор Турбо Паскаля, неизбежно вносит в машинный код некоторую избыточность, уменьшающую скорость счета и увеличивающую затраты памяти. С другой стороны, программирование на уровне машинных инструкций — чрезвычайно хлопотное занятие и не может сравниться по скорости разработки программ с программированием на Паскале — в этом заключается главный недостаток ассемблера.

Чтобы использовать средства ассемблера, необходимо ясно представлять себе детали архитектуры микропроцессоров Intel 80x86. К этому семейству относятся микропроцессоры:

8086 — 16-разрядный микропроцессор, используемый в ПК *IBM PC/XT*;

8088 — аналог 8086, отличается от него только взаимодействием с памятью: 8086 может обмениваться с памятью как байтами, так и 16-разрядными словами, в то время как 8088 — только байтами;

80286 — улучшенный вариант 8086, используемый в ПК *IBM AT*; может работать в двух режимах: в реальном режиме, полностью эмулирующем работу МП 8086, и в защищенном режиме, в котором способен адресовать память до 16 Мбайт (в реальном — до 1 Мбайт);

80386 — 32-разрядный вариант 80286; способен адресовать до 4 Гбайт;

80486 — комбинация 80386/80387, т.е. имеет внутреннюю подсистему реализации операций с плавающей точкой;

80586 (Pentium) — имеет ряд усовершенствований, обеспечивающих ему увеличение производительности в 2...3 раза по сравнению с 80486, в том числе возможность обрабатывать 64-разрядные числа.

Микропроцессоры этого семейства наращивают свои возможности в перечисленном порядке, но строго совместимы от младших моделей к старшим: все, что может 8086/8088, реализует и *Pentium*, но не наоборот. Ниже обсуждается архитектура (внутреннее устройство, способы адресации и система команд) МП 8086/8088.

12.1.1. Регистры

В МП 8086/8088 имеется 14 регистров. В функциональном отношении они делятся на группы:

- регистры общего назначения (*AX, BX, CX, DX*); предназначены для хранения операндов и выполнения основных команд; любой из них может использоваться как совокупность двух независимых друг от друга 8-разрядных регистров: старшего байта регистра (*AH, BH, CH, DH*) и младшего байта (*AL, BL, CL, DL*); например, *AX* состоит из *AH* и *AL*;
- сегментные регистры (*CS, DS, SS, ES*); используются для указания сегмента при адресации памяти;
- регистры-указатели (*SP, BP, IP*); используются для указания смещения при адресации памяти;
- индексные регистры (*SI, DI*); применяются для индексной адресации;
- регистр флагов; используется для хранения признаков состояния процессора.

Внутри одной и той же функциональной группы регистры используются различным образом. Ниже описывается специфика использования регистров.

Регистр *AX*. Является основным сумматором. Используется во всех арифметических операциях (сложить, умножить и т.п.). Только с помощью *AX* и его полурегистров *AH/AL* возможен обмен данными с портами ввода/вывода.

Регистр *BX*. Используется как сумматор в арифметических операциях, а также как базовый регистр при индексной адресации.

Регистр *CX*. В основном используется как счетчик при выполнении операций повторения и сдвига. Может также участвовать в арифметических операциях.

Регистр *DX*. Используется как регистр данных в операциях ввода/вывода, а также как сумматор при обработке длинных целых чисел (32-разрядных).

Регистр *CS*. Содержит номер сегмента памяти (сегмента кода), в котором располагается текущая машинная инструкция. Для получения полного адреса следующей команды его содержимое сдвигается влево на 4 разряда и складывается с регистром-указателем *IP*. Содержимое *CS* автоматически изменяется в командах дальнего (межсегментного) перехода и вызова процедур.

Регистр IP. Определяет смещение относительно начала сегмента кода CS очередной исполняемой машинной инструкции. Содержимое IP автоматически изменяется в ходе исполнения инструкции, обеспечивая правильный порядок выборки команд из памяти.

Регистр DS. Содержит номер сегмента памяти (сегмента данных), в котором располагаются данные (константы и переменные). Все глобальные переменные и типизированные константы программы Турбо Паскаля всегда располагаются в единственном сегменте, адресуемом этим регистром.

Регистр SS. Содержит номер сегмента стека. Стек — это участок автоадресуемой памяти, предназначенный для временного хранения операндов. С помощью стека Турбо Паскаль организует обмен данными между программой и процедурами, кроме того, в нем он размещает все локальные переменные (т.е. переменные, объявленные внутри процедуры). Память стека используется по правилу «последним пришел — первым ушел»: самый последний помещенный в стек операнд будет первым извлекаться из него.

Регистр SP. Указывает на вершину стека, т.е. совместно с регистром SS адресует ячейку памяти, куда будет помещаться операнд или откуда он будет извлекаться. Содержимое этого регистра автоматически уменьшается после размещения в стеке очередного операнда и увеличивается после извлечения операнда из стека.

Регистр BP. Так называемый указатель базы. Облегчает создание и использование локального стека (т.е. стека для использования внутри процедуры).

Регистр ES. Дополнительный сегментный регистр ES используется для межсегментного обмена данными и в некоторых строковых операциях.

Регистр SI. Определяет адрес источника информации при индексной адресации данных (например, при обработке массивов). Обычно используется в паре с регистром DS.

Регистр DI. В паре с регистром ES определяет приемник информации при межсегментном обмене данными.

Регистр флагов. Отдельные разряды (биты) этого регистра имеют следующее назначение.

Флаг переноса CF. Содержит 1, если произошел перенос единицы при сложении или заем единицы при вычитании. Используется также в циклических операциях и операциях сравнения.

Флаг четности PF. Содержит 1, если в результате операции получено число с четным количеством значащих разрядов, т.е. дополняет результат до нечета — используется в операциях обмена для контроля данных.

Флаг внешнего переноса AF. Контролирует перенос из 3-го бита данных. Полезен при операциях над упакованными десятичными числами.

Флаг нуля ZF. Равен 1, если в результате операции получен ноль, и равен 0 в противном случае.

Флаг знака SF. Равен 1, если в результате операции получено отрицательное число (с единицей в старшем разряде).

Флаг трассировки *TF*. Равен 1, если программа выполняется по шагам, с передачей управления после каждой выполненной команды по прерыванию с вектором 1.

Флаг прерываний *IF*. Содержит 1, если микропроцессору разрешена обработка прерываний.

Флаг направления *DF*. Управляет направлением передачи данных: если он содержит 0, то после каждой индексной операции содержимое индексных регистров увеличивается на 1, в противном случае — уменьшается на 1.

Флаг переполнения *OF*. Устанавливается в единицу, если в результате операции получено число, выходящее за разрядную сетку микропроцессора.

12.1.2. Адресация

В архитектуре МП 8086/8088 адрес любого байта задается двумя 16-битовыми словами — сегментом и смещением. При формировании 20-разрядного полного адреса, необходимого для адресации в пределах 1 Мбайт, сегмент сдвигается влево на 4 разряда (умножается на 16) и складывается со смещением. Поскольку емкость 16-разрядного смещения составляет 65536 значений, в пределах одного сегмента можно адресовать до 64 Кбайт.

Архитектура МП позволяет использовать семь различных способов адресации.

Регистровая

Извлекает операнд из регистра или помещает его в регистр. Примеры:

<code>mov ax, bx</code>	{Извлекаем из <i>BX</i> и помещаем в <i>AX</i> }
<code>add cx, ax</code>	{Содержимое <i>AX</i> прибавляем к <i>CX</i> }
<code>push cx</code>	{Заталкиваем в стек содержимое <i>CX</i> }

Непосредственная

Операнд (8- или 16-разрядная константа) содержится непосредственно в теле команды. Примеры:

<code>mov ax, 100</code>	{Загружаем в <i>AX</i> значение 100}
<code>add ax, 5</code>	{К содержимому <i>AX</i> прибавляем 5}
<code>mov cx, \$FFFF</code>	{Помещаем в <i>CX</i> значение 65535}

Прямая

Смещение операнда задается в теле программы и складывается с регистром *DS*; например:

var

X: Word;
B: Byte;

.....

<code>mov ax, X</code>	{Пересылаем значение переменной <i>X</i> регистр <i>AX</i> }
<code>add ah, B</code>	{К содержимому регистра <i>AH</i> прибавляем

значение переменной *B*}

```
mov X, ax {Пересылаем содержимое регистра AX
           в область памяти переменной X}
```

Косвенная регистровая

Исполнительный адрес операнда (точнее, его смещение) содержится в одном из регистров *BX*, *BP*, *SI* или *DI*. Для указания косвенной адресации этот регистр должен заключаться в квадратные скобки, например:

```
mov ax, [bx] {Содержимое 16-разрядного слова, хранящегося
              в памяти по адресу DS:BX,
              пересылаем в регистр AX};
```

Каждый из регистров *BX...DI* по умолчанию работает со своим сегментным регистром:

DS:BX, *SS:BP*, *DS:SI*, *ES:DI*

Допускается явное указание сегментного регистра, если он отличается от умалчиваемого, например:

```
mov ax, es:[bx]
```

Адресация по базе

Базовый регистр *BX* (или *BP*) содержит базу (адрес начала некоторого фрагмента памяти), относительно которой ассемблер вычисляет смещение, например:

```
mov ax, [bx]+10 {Загружаем в AX 10-й по счету
                 байт от начала базы памяти
                 по адресу DS:BX};
```

Индексная адресация

Один из индексных регистров *SI* или *DI* указывает положение элемента относительно начала некоторой области памяти. Пусть, например, *AOB* — имя массива значений типа *Byte*. Тогда можно использовать такие фрагменты:

```
mov si, 15      {Помещаем в SI константу 15}
mov ah, AOB[si] {Пересылаем в AH 16-й по
                порядку байт от начала массива}

mov si, 0
mov AOB[si], ah {Пересылаем полученное в
                самый первый элемент массива}
```

Адресация по базе с индексированием

Вариант индексной адресации для случая, когда индексируемая область памяти задается своей базой. Например:

```
mov ax, [bx][si]
```

Этот тип адресации удобен при обработке двумерных массивов. Если, например, *AOB* есть массив из 10x10 байт вида

```
var
  AOB: array [0..9,0..9] of Byte;
```

то для доступа к элементу *AOB* [2,3] можно использовать такой фрагмент

```
mov bx,20           {База строки 2}
mov si,2            {Номер 3-го элемента}
mov ax,AOB[bx][si]  {Доступ к элементу}
```

12.1.3. Система команд

В приводимых ниже таблицах указывается мнемоника всех допустимых инструкций для МП 8086/8088. Для удобства пользования все команды разбиты на 6 функциональных групп – пересылки данных, арифметические, битовые, строковые, передачи управления, прерываний. Внутри каждой группы команды объединяются в подгруппы по общим дополнительным признакам.

Детальный анализ всех команд МП 8086/8088 занял бы слишком много места, поэтому в идущих за таблицами пояснениях рассматриваются лишь наиболее популярные команды. Исчерпывающее описание всех команд Вы найдете в [1], [20].

Команды пересылки данных

Мнемоника	Формат	Пояснение
Команды общего назначения		
MOV	MOV приемник, источник	Переслать значение
PUSH	PUSH источник	Поместить в стек
POP	POP приемник	Извлечь из стека
XCHG	XCHG приемник, источник	Обменять значения
XLAT	XLAT таблица	Загрузить в AL байт из таблицы
Команды ввода-вывода		
IN	IN аккумулятор, порт	Читать из порта
OUT	OUT порт, аккумулятор	Записать в порт
Команды пересылки адреса		
LEA	LEA регистр16, память16	Загрузить исполнительный адрес
LDS	LDS регистр16, память32	Загрузить в DS:регистр16 полный адрес
LES	LES регистр16, память32	Загрузить в ES:регистр16 полный адрес
Команды пересылки флагов		
LAHF	LAHF	Загрузить флаги в AH
SAHF	SAHF	Установить флаги из AH
PUSHF	PUSHF	Поместить флаги в стек
POPF	POPF	Извлечь флаги из стека

Одна из наиболее часто используемых команд – MOV позволяет в защищенном режиме переслать байт или слово из регистра в регистр из

памяти в регистр или из регистра в память. Тип пересылаемых данных (байт или слово) определяется регистром, участвующим в пересылке. Ниже приводятся примеры использования команды:

```
mov ax,Table    {Пересылка слова из памяти в AX}
mov Table,ah    {Пересылка байта из AH в память}
mov ds,ax       {Пересылка в сегмент данных}
mov es:[bx],ax  {Пересылка слова в память: базовая
                  адресация с заменой сегмента}
mov ch,-17      {Переслать константу в регистр}
mov Table,$FF   {Переслать константу в память}
```

С помощью MOV нельзя пересылать:

- из памяти в память, например, вместо

```
mov Mem1,Mem2
```

следует использовать

```
mov ax,Mem2
mov Mem1,ax
```

- константу или переменную в DS, например, нельзя

```
mov DS,Data_Seg
```

нужно:

```
mov ax,Data_Seg
mov ds,ax
```

- один сегментный регистр в другой, например, нельзя

```
mov es,ds
```

но можно

```
mov ax,ds
mov es,ax
```

- в регистр CS; значение этого регистра (сегмента кода) автоматически меняется при выполнении дальних команд CALL и JMP; кроме того, он загружается из стека при выполнении команды RETF (выход из дальней процедуры).

Для временного сохранения регистров и данных, а также для обмена значениями между регистрами широко используются стековые команды PUSH и POP. Каждая из них работает со словом, т.е. в стек нельзя поместить или извлечь из него одиночный байт. При выполнении PUSH в начале уменьшается на 2 содержимое указателя SP, а затем операнд помещается по адресу SS: SP. При извлечении из стека сначала читается память по адресу SS: SP, а затем SP увеличивается на 2. Таким образом, при заполнении указатель вершины стека SP смещается к младшим адресам, а при освобождении — к старшим. При работе со стеком следует помнить о специфике использования стековой памяти («последним пришел — первым

ушел»), а также о том, что эта память интенсивно используется при вызове процедур, т.е. состояние стека к моменту выхода из процедуры должно быть строго согласовано с дальнейшей работой программы. Первое условие определяет порядок извлечения данных из стека — он должен быть обратным порядку, в котором эти данные помещались в стек. Второе условие фактически означает, что после выхода из процедуры указатель *SP* должен содержать то же смещение, что и к моменту входа в нее. Иными словами, процедура не должна «забыть» в стеке лишнее слово или взять из него больше нужного.

Команда загрузки адреса *LEA* загружает в регистр адрес (смещение) нужного участка памяти. Этого же можно достичь с помощью зарезервированного слова *OFFSET*, стоящего перед именем переменной. Например:

```
var
    X: Word;
    .....
asm
    mov ax, OFFSET X {Загружаем смещение X в AX}
    lea ax, X         {То же действие}
end;
```

Разница состоит в том, что в случае команды *LEA* разрешается использовать индексную адресацию, что особенно удобно при пересылке массивов данных.

Две другие команды адресной загрузки — *LDS* и *LES* загружают первое 16-разрядное слово из источника в регистр-приемник, а затем следующее слово — в регистр *DS* или *ES*, т.е. они рассчитаны на загрузку полного адреса операнда (сегмента и смещения).

Арифметические команды

Мнемоника	Формат	Комментарий
Команды сложения		
ADD	ADD приемник, источник	Сложить
ADC	ADC приемник, источник	Сложить, добавить перенос
AAA	AAA	Скорректировать сложение для таблицы ASCII
DAA	DAA	Скорректировать сложение для двоично-десятичных чисел
INC	INC приемник	Увеличить на единицу
Команды вычитания		
SUB	SUB приемник, источник	Вычесть
SBB	SBB приемник, источник	Вычесть с заемом
AAS	AAS	Скорректировать вычитание для таблицы ASCII
DAS	DAS	Скорректировать вычитание для двоично-десятичных чисел
DEC	DEC приемник	Уменьшить на единицу
NEG	NEG приемник	Обратить знак

CMP	CMP приемник, источник	Сравнить
Команды умножения		
MUL	MUL источник	Умножить без знака
IMUL	IMUL источник	Умножить со знаком
AAM	AAM	Скорректировать умножение для таблицы ASCII
Команды деления		
DIV	DIV источник	Делить без знака
IDIV	IDIV источник	Делить со знаком
AAD	AAD	Скорректировать деление для таблицы ASCII
Команды расширения знака		
CBW	CBW	Преобразовать байт в слово
CWD	CWD	Преобразовать слово в двойное слово

При использовании арифметических команд следует помнить о том, что МП может обрабатывать знаковые числа, числа без знака, а также двоично-десятичные числа. В беззнаковых числах для представления значения используются все биты, т.е. они эквивалентны типам *Byte* и *Word*, в то время как знаковые числа в старшем разряде хранят знак числа и эквивалентны типам *ShortInt* и *Integer*. Двоично-десятичные числа используют по 4 бита для каждого десятичного разряда и могут быть упакованными или неупакованными. В первом случае один байт хранит 2 десятичные цифры (старшая — в старшем полубайте), во втором — только одну (старший полубайт не используется). Основные арифметические команды МП (*ADD*, *SUB*, *MUL*, *DIV*) не учитывают двоично-десятичную форму представления чисел, поэтому в архитектуру МП включены команды коррекции результата.

Битовые команды

Мнемоника	Формат	Комментарий
Логические команды		
AND	AND приемник, источник	Выполнить AND
OR	OR приемник, источник	Выполнить OR
XOR	XOR приемник, источник	Выполнить XOR
NOT	NOT приемник	Выполнить NOT
TEST	TEST приемник, источник	Проверить
Сдвиговые команды		
SAL/SHL	SAL приемник, счетчик	Сдвинуть влево
SAR/SHR	SAR приемник, счетчик	Сдвинуть вправо
ROL	ROL приемник, счетчик	Сдвинуть влево циклически
ROR	ROR приемник, счетчик	Сдвинуть вправо циклически
RCL	RCL приемник, счетчик	Сдвинуть влево с переносом
RCR	RCR приемник, счетчик	Сдвинуть вправо с переносом

Битовые команды используются при исчислении логических выражений, а также в тех случаях, когда необходимо изменить отдельные разряды

операнда. Логические команды *AND*, *OR*, *XOR* и *NOT* эквивалентны соответствующим операциям Турбо Паскаля в случае, когда операндами являются целочисленные выражения. Команда *TEST* выполняет целочисленную операцию поразрядного суммирования *AND*, но не изменяет значения операндов, а лишь устанавливает флаги в соответствии со значением результата сравнения: обнуляет *CF* и *OF*, изменяет *PF*, *ZF*, *SF* и не меняет *AF* (флаг *ZF* установится в 1 в том случае, когда оба операнда содержат по единице хотя бы в одном соответствующем разряде). Команды сдвига *SHL/SHR* эквивалентны одноименным операциям Турбо Паскаля и отличаются от команд циклического сдвига *ROL/ROR* тем, что вытесненные в ходе их выполнения значащие разряды теряются, в то время как при циклическом сдвиге эти разряды появляются «с другой стороны». Например, если выполнить фрагмент

```
mov al,1    {Загружаем в AL единицу}
shr al,1    {Сдвигаем вправо на 1 разряд}
```

регистр *AL* будет содержать 0 (вытесненная вправо единица будет помещена в *CF*), в то время как после замены команды *SHR* на *ROR* в нем будет значение \$80=128 (вытесненная единица будет помещена в старший бит регистра).

Заметим, что счетчиком в командах сдвига может быть цифра 1 или количество сдвигов, указываемое в регистре *CL*.

Команды передачи управления

Мнемоника	Формат	Комментарий
Безусловные переходы		
<i>CALL</i>	<i>CALL</i> имя	Войти в процедуру
<i>RET</i>	<i>RET</i> [число_параметров]	Вернуться из процедуры
<i>JUMP</i>	<i>JUMP</i> имя	Перейти
Условные переходы		
<i>JA/JNBE</i>	<i>JA</i> близкая_метка	Перейти, если выше (после сравнения беззнаковых операндов)
<i>JAE/JNB</i>	<i>JAЕ</i> близкая_метка	Перейти, если выше или равно
<i>JB/JBAE/JC</i>	<i>JB</i> близкая_метка	Перейти, если ниже
<i>JBE/JNA</i>	<i>JBE</i> близкая_метка	Перейти, если ниже или равно
<i>JCXZ</i>	<i>JCXZ</i> близкая_метка	Перейти, если <i>CX</i> =0
<i>JE/JZ</i>	<i>JE</i> близкая_метка	Перейти, если равно
<i>JG/JNLE</i>	<i>JG</i> близкая_метка	Перейти, если больше (после сравнения знаковых операндов)
<i>JGE/JNL</i>	<i>LGE</i> близкая_метка	Перейти, если больше или равно
<i>JL/JNGE</i>	<i>JL</i> близкая_метка	Перейти, если меньше
<i>JLE/JNG</i>	<i>JLE</i> близкая_метка	Перейти, если меньше или равно
<i>JNC</i>	<i>JNC</i> близкая_метка	Перейти, если нет переноса
<i>JNE/JNZ</i>	<i>JNE</i> близкая_метка	Перейти, если не равно
<i>JNO</i>	<i>JNO</i> близкая_метка	Перейти, если нет переполнения
<i>JNP/JPO</i>	<i>JNP</i> близкая_метка	Перейти, если нечетный
<i>JO</i>	<i>JO</i> близкая_метка	Перейти, если перенос
<i>JP/JPE</i>	<i>JP</i> близкая_метка	Перейти, если четный
<i>JS</i>	<i>JS</i> близкая_метка	Перейти, если отрицательный

Команды управления циклами

LOOP	LOOP близкая_метка	Повторить цикл
LOOPE/LOOPZ	LOOPE близкая_метка	Повторять, пока равны
LOOPNE/LOOPNZ	LOOPNE близкая_метка	Повторять, пока не равны

Команды безусловных переходов *CALL*, *RET*, *JMP* могут использовать дальнюю или ближнюю модель памяти, в то время как команды условных переходов — только малую (в пределах $-128...+127$ байтов). При дальней модели памяти (устанавливается опцией *Options/Compiler/Force far calls* среды Турбо Паскаля или директивой компилятора *{F+}*) осуществляется как внутрисегментная, так и межсегментная передача управления, при ближней — только внутрисегментная.

Инструкция *CALL* работает следующим образом. Вначале адрес следующей за *CALL* инструкции (адрес возврата) помещается в стек, затем в регистр *IP* (или в пару *CS:IP*) помещается адрес точки входа в процедуру, таким образом сразу за командой *CALL* будет исполняться уже первая команда процедуры. Оба адреса (точки входа и возврата) будут 16-битовыми смещениями для внутрисегментного вызова или 32-битовыми полными адресами — для межсегментного. Все процедуры (функции) Паскаля, оттранслированные в режиме *{F+}* или содержащие зарезервированное слово *FAR* в заголовке, должны вызываться как дальние. Для этого за инструкцией *CALL* следует указать модель памяти:

```

Procedure MyProc; Far;
.....
asm
call FAR MyProc    {Вызов дальней процедуры}
...
end;
```

Таким же способом должны вызываться все библиотечные подпрограммы (т.е. объявленные в интерфейсных частях модулей). При дальнем вызове в стек сначала заталкивается содержимое сегмента кода *CS*, а уже затем — смещение возврата.

При выходе из дальней процедуры команда *RET* извлекает из стека оба 16-разрядных слова и помещает первое в *IP*, а второе в *CS*, а при выходе из ближней извлекает из стека только смещение и помещает его в *IP*.

Команды условных переходов способны передавать управление на метку, расположенную в пределах ближайших плюс-минус 128 байт от самой команды. Если нужно передать управление на метку, расположенную дальше в том же сегменте, или на метку в другом сегменте, сразу за командой условной передачи располагают безусловную команду *JMP* или *CAL*, например:

```

cmp ax,0           {Проверяем AX}
jne @NotZero       {AX=0 ?}
jmp IsZero         {Да - переходим на дальнюю метку}
.....            {Нет - продолжаем работу}
.....
```

В таблице термин «выше/ниже» используется применительно к сравнению беззнаковых операндов, а «больше/меньше» — знаковых.

Поскольку условные переходы реализуют ветвление программы на основе проверки флагов, обычно непосредственно перед ними располагаются команды, изменяющие эти флаги, чаще всего — команда сравнения *CMR*. Ниже показаны комбинации *CMR*—условный_переход для различных соотношений приемника и источника (первого и второго операнда) команды *CMR*:

Условие	Для беззнаковых чисел	Для чисел со знаками
Приемник больше источника	JA	JG
Приемник и источник равны	JE	JE
Приемник меньше источника	JB	JL
Приемник не меньше источника	JAЕ	JGE
Приемник не больше источника	JBE	JLE
Приемник и источник не равны	JNE	JNE

Например:

```

cmp ax,5          {AX>5 ?}
ja @Above5        {Да, больше - переходим}
cmp bx,-3         {BX<=-3 ?}
jle @LessM3       {Да, меньше или равен}

```

Команды *LOOP/LOOPE/LOOPNE* служат для организации циклов. Все они используют содержимое регистра *CX* как счетчик числа повторений. Команда *LOOP* уменьшает *CX* на единицу и передает управление на метку начала цикла, если содержимое этого регистра отлично от нуля. Команды *LOOPE/LOOPNE* также уменьшают счетчик *CX*, но передают управление в начало цикла при совместном условии установки (или сброса) флага *ZF* и неравенства нулю счетчика *CX*.

Вот как, например, можно отыскать нулевой байт в массиве *AOB*:

```

var
  AOB: array [1..1000] of Byte;
.....
asm
  mov cx,1000          {Иницилируем счетчик CX}
  lea bx,AOB           {Помещаем адрес AOB в BX}
  dec bx               {Готовим цикл}
  {Здесь начало цикла проверки}
@@Test: inc bx         {Адрес очередного байта}
  cmp BYTE PTR [bx],0  {Проверяем байт}
  loopne @Test         {Замыкаем цикл}
  jnz @NotZero         {Если не найден нулевой байт}
  .....              {Нашли нулевой байт}
end;

```

Строковые команды

Мнемоника	Формат	Комментарий
Пересылка строк		
MOVSB	MOVSB	Пересылать байты
MOVSW	MOVSW	Пересылать слова
Сравнение строк		
CMPSB	CMPSB	Сравнивать байты
CMPSW	CMPSW	Сравнивать слова
Сканирование		
SCASB	SCASB	Искать байт
SCASW	SCASW	Искать слово
Загрузка и сохранение		
LODSB	LODSB	Загружать байты
LODSW	LODSW	Загружать слова
STOSB	STOSB	Сохранять байты
STOSW	STOSW	Сохранять слова

Строковые команды рассчитаны на обработку строк. Замечу, что термин «строка» здесь отнюдь не эквивалентен аналогичному термину Турбо Паскаля и означает произвольную цепочку байт или слов длиной до 64 Кбайт. Эти команды оперируют пятью примитивами, каждый из которых обрабатывает лишь один байт или одно слово за раз. Перед примитивом обычно указывается префикс повторения *REP/REPE/REPNE*, заставляющий выполняться примитив до тех пор, пока не обнулится счетчик повторений *CX* или не будет нарушено соответствующее условие.

При использовании строковых команд важно помнить два обстоятельства. Во-первых, эти команды всегда берут адрес строки-источника из пары *DS:SI*, а строки-приемника — из пары *ES:DI*. Таким образом, перед исполнением строковой команды необходимо инициализировать сегментные регистры нужным образом. Во-вторых, строковые команды используют индексную адресацию с автоматическим изменением смещения в *SI/DI* после однократного исполнения примитива. Содержимое этих регистров изменяется на 1 при обработке байтов и на 2 при обработке слов, причем наращивается, если флаг направления *DF* сброшен, и уменьшается, если он равен 1.

Вот как можно осуществить пересылку массива *A* в массив *B*:

```
var
  A,B: array [1..250] of Integer;
.....
asm
  lea si, A      {Смещение A - в SI (источник)}
  push ds
  pop es         {Инициализируем ES := DS}
  lea di, B      {Смещение B - в DI (приемник)}
  mov cx, 250    {Счетчик переноса}
  cld            {Направление переноса - наращивать}
```

```
rep movsw      {Переносим 500 байт}
end;
```

В программе на Турбо Паскале регистр *DS* всегда содержит сегмент данных, поэтому инициировать его необязательно. Что касается регистра дополнительного сегмента *ES*, такого правила нет, и хотя в большинстве случаев он также ссылается на сегмент данных, рекомендуется проводить его инициацию перед использованием строковой команды (см. выше ко — манды *push ds*, *pop es*).

Команды прерываний

Мнемоника	Формат	Комментарий
INT	INT номер	Выполнить прерывание
INTO	INTO	Выполнить прерывание по переполнению
IRET	IRET	Вернуться из прерывания

Выполнение прерываний во многом напоминает косвенный вызов дальнейшей процедуры. По команде *INT* (*INTO*) в стек помещается регистр флагов, сегмент *CS* и указатель *IP*, а новые значения этих регистров берутся из 4-байтного вектора прерывания, соответствующего номеру прерывания в команде *INT*, или из вектора 4 — для команды *INTO*. Таким образом, единственным отличием от команды *CALL* является то, что в стек предварительно заносится регистр флагов. Следует, правда, оговориться: перед передачей управления программе обработки прерывания микропроцессор сбрасывает флаги трассировки *TF* и прерываний *IF*; сброс *TF* необходим для обеспечения нормальной работы отладчиков, использующих прерывание по вектору 1 или 4, сброс *IF* блокирует вмешательство других процессов в ход обработки прерывания.

Команда *INTO* представляет собой условное прерывание и выполняется, если в этот момент взведен флаг переполнения *OF*. Команда *IRET* реализует правильный выход из программы обработки прерывания: она считывает из стека 3 двухбайтных слова и помещает их в регистры *IP*, *CS* и регистр флагов.

Команды управления

Мнемоника	Формат	Комментарий
Управление флагами		
STC	STC	Установить перенос
CLC	CLC	Очистить перенос
CMC	CMC	Инвертировать CF
STD	STD	Установить направление
CLD	CLD	Очистить направление
STI	STI	Разрешить прерывания
CLI	CLI	Запретить прерывания
Внешняя синхронизация		
HLT	HLT	Остановить вычисления
WAIT	WAIT	Ждать активности на

		шине
ESC	ESC код, источник	Передать команду
LOCK	LOCK	Захватить шину
Пустая команда		
NOP	NOP	Нет операции

Команды внешней синхронизации работают следующим образом.

HALT переводит МП в состояние останова, из которого его можно вывести только при перезагрузке системы или при наступлении немаскируемого прерывания.

WAIT заставляет МП выполнять холостой режим работы и каждые 5 тактов проверять уровень сигнала на входной шине: пока на этой шине нет сигнала активности, процессор выполняет **WAIT**, но как только шина активизируется, он продолжит исполнение программы. Эта инструкция обычно используется для ожидания сигнала обслуживания (прерывания) высокоприоритетного устройства типа контроллера прямого доступа к памяти.

Команда **ESC** используется для передачи указанного в ней операнда на шину данных. Тем самым обеспечивается возможность передачи команд другим процессорам. Эта команда чаще всего используется для управления работой арифметического сопроцессора. В этом случае код представляет собой код команды сопроцессора, а источник — используемый в этой команде операнд.

Команда **LOCK** фактически представляет собой однобайтовый префикс, который можно использовать совместно с любой другой командой микропроцессора. По этой команде МП активизирует одноименный сигнал на своей шине, что исключает возможность использования этой шины любым другим внешним устройством (процессором).

12.2. СПЕЦИФИКА ВСТРОЕННОГО АССЕМБЛЕРА

Приведенное выше общее описание архитектуры МП 8086/8088 является базовым для любого ассемблера, в том числе и для встроенного ассемблера Турбо Паскаля. Однако ассемблеры содержат массу дополнительных возможностей, облегчающих разработку готовых к работе программ. Эти возможности отражаются в директивах и макрокомандах ассемблера. Встроенный ассемблер не предназначен для написания законченных программ, поэтому в нем отсутствуют макрокоманды и директивы. Главной особенностью встроенного ассемблера является практически полное отсутствие в нем средств описания переменных и данных, т.к. эти объекты описываются средствами Турбо Паскаля.

12.2.1. Оператор ASM

Зарезервированное слово **ASM** открывает доступ к средствам встроенного ассемблера. Этот оператор может располагаться только внутри исполняемой части программы (подпрограммы). Область действия оператора **ASM** ограничивается ближайшим по тексту зарезервированным

словом **END**. Таким образом, структура любого ассемблерного оператора такова:

asm

<Одна или несколько команд встроенного ассемблера>

end;

С точки зрения Турбо Паскаля пара **asm... end** считается операторными скобками, ограничивающими единственный оператор Паскаля, например:

```
if X>10 then
  asm
    .....
  end
else
  .....;
for k := 1 to 5 do
  asm
    .....
end;
```

Тело ассемблерного оператора **asm... end** может быть пустым или содержать несколько ассемблерных команд. Каждая ассемблерная команда должна располагаться на отдельной строке или отделяться от следующей за ней команды символом «;». Ниже приводятся два разных способа написания одной и той же последовательности ассемблерных команд:

```
asm
  mov ah,0; int $16; mov ChCode, al; mov ScCode, ah
end;
asm
  mov ah,0
  int $16
  mov ChCode,al
  mov ScCode,ah
end;
```

В конце строки, содержащей единственную ассемблерную команду, или между двумя командами, располагающимися на одной строке, разрешается вставлять комментарий, который должен оформляться по обычным правилам Турбо Паскаля, т.е. ограничиваться символами «{», «}» или «(», «)». Таким образом, комментарии разрешены между ассемблерными командами, но не внутри них. Например, такой оператор будет правильным:

```
asm
  {Инициализируем регистры}
  lea si,X; push ds;
  pop es;           {ES := DS}
  lea di,Y; mov cx,100
  cld               {Перенос - вперед}
  rep {Выполняем Y := X} movsw
```

{Здесь нет ошибки – комментарий можно
вставлять между префиксом и командой}

end;

а такой – неправильным:

asm

{Готовим регистры}

lea si,X; push ds;

pop {ES:=DS} es;

{Ошибка! Комментарий разорвал

мнемонику команды и ее операнд}

lea di,Y; mov cx,100 {и направление} cld

{Комментарий является разделителем команд,

поэтому перед ним можно не ставить ";"}

rep movsw

end;

В пределах ассемблерного оператора допускаются любые команды, но Турбо Паскаль требует выполнения следующего соглашения:

В начале ассемблерного оператора регистр DS содержит сегмент кода, SS – сегмент стека, BP – текущий стек, SP указывает на вершину стека.

Все эти регистры должны иметь точно такие же значения к моменту завершения работы ассемблерного оператора.

Программист не должен делать каких-либо предположений о содержимом остальных регистров, и эти регистры могут иметь произвольное значение после завершения работы ассемблерного оператора. Исключением является случай ассемблерной функции, которая должна использовать некоторые регистры для возврата своего значения (см. п.12.2.3).

12.2.2. Синтаксис ассемблерных команд

Здесь и далее ассемблерными командами называются команды на языке встроенного ассемблера, вставляемые в тело ассемблерного оператора *asm... end*.

Структура ассемблерной команды такова:

[Метка] [Префикс] [Код [Операнд [,Операнд]]]

В квадратных скобках указываются необязательные элементы структуры.

Метки

Любой команде ассемблерного оператора может предшествовать одна или несколько меток. В ассемблере используется два типа меток: глобальные и локальные. Глобальные метки – это обычные метки Турбо Паскаля. Они объявляются в разделе описаний после зарезервированного слова **Label**. С помощью глобальной метки можно передать управление в тело ассемблерного оператора оператором **GOTO**. Например:

```

Label
  AltEnt;
begin
  .....
  Goto AltEnt;      {Передаем управление внутрь
                    ассемблерного оператора}
  .....
asm
  .....
AltEnt:              {Сюда можно передать управление извне}
  .....
end;

```

Локальные метки объявляются непосредственно в теле ассемблерного оператора. Эти метки обязаны начинаться символом «@». Поскольку этот символ нельзя использовать в именах Турбо Паскаля, он позволяет отличать локальную метку от глобальной. Локальная метка не известна нигде вне ассемблерного оператора, поэтому на нее нельзя передать управление оператором *GOTO*. По этой же причине в разных ассемблерных операторах можно использовать одноименные локальные метки.

Префиксы

Встроенный ассемблер поддерживает следующие префиксы команд:

LOCK	Захват шины
REP/REPE/REPNE	Повтор строковой команды
REPZ/REPNZ	Синоним REPE/REPNE
SEGCS	Перекрытие CS
SEGDS	Перекрытие DS
SEGSS	Перекрытие SS
SEGES	Перекрытие ES

Префиксы *LOCK/REP/REPE/REPNE* описаны в п.12.1.3. Префиксы *SEGxx* определяют сегментный регистр, который должен использоваться вместо умалчиваемого, и распространяются только на следующие за ними ассемблерные команды.

Если префикс указан без кода инструкции, он распространяет свое действие на следующую ассемблерную команду.

Код инструкции очень редко имеет более одного префикса и никогда — более трех: допускается следующая последовательность

LOCK SEGxx REPxx

Замечу, что если при обработке строковой команды произошло аппаратное прерывание, МП 8086/8088 «забывает» префиксы *LOCK* и *SEGxx*, которые, возможно, определены в той же команде, так что использовать сложные префиксные конструкции не рекомендуется.

Коды инструкций

Встроенный ассемблер поддерживает мнемонику всех команд, перечисленных в п.12.1.3. Кроме того, в ассемблерных командах может использоваться мнемоника инструкций процессора 8087, а также команды процессоров 80286/80287. Замечу, что инструкции 8087 допустимы только при активном состоянии $\{ \$N+ \}$, 80286 — при $\{ \$G+ \}$, а 80287 — в случае $\{ \$G+, N+ \}$.

Операнды

Операндами встроенного ассемблера могут быть выражения, состоящие из комбинации регистров, констант, имен и символов операций.

Регистры

Во встроенном ассемблере используется мнемоника регистров, указанная в п.12.1.1, а также имя *ST* для ссылки на регистры арифметического сопроцессора.

Константы

Ассемблер поддерживает строковые и числовые константы.

Строковые константы заключаются в апострофы или кавычки. Если константа объявлена с помощью кавычек, внутри нее символ апострофа рассматривается наравне с другими символами, т.е. не считается ограничителем константы, точно так же внутри константы, обрамленной апострофами, не считается ограничителем символ кавычки. Если внутри константы необходимо указать ограничивающий ее символ, он удваивается. Примеры:

```
'Строковая константа'
"Это - тоже строковая константа"
'Символ ' ' не считается ограничителем'
'внутри строки, обрамленной кавычками "..."
```

Числовые константы могут быть только целыми и их значение не может превосходить емкости двойного слова, т.е. должно быть внутри диапазона

- 2 147 483 648 ... + 4 294 967 295.

По умолчанию при записи числовых констант используется десятичная нотация, но ассемблер поддерживает также двоичные, восьмеричные и шестнадцатеричные константы. Двоичная константа составляется как комбинация единиц и нулей, заканчивающаяся символом *B* (от *Binary* — двоичный); при записи восьмеричной константы используются символы 0...7, а в ее конце ставится символ *O* (*Octal* — восьмеричный); шестнадцатеричная константа записывается по правилам Турбо Паскаля (начинается с символа *#*) либо по правилам Турбо Ассемблера: начинается с цифры, в конце ставится символ *H* (от *Hexadecimal* — шестнадцатеричный).

Имена

Локальные метки — это единственные имена, которые разрешается определять внутри ассемблерного оператора. Имена остальных объектов

программы — констант, переменных, подпрограмм — должны определяться только с помощью средств Турбо Паскаля.

Область определения имен подчиняется тем же правилам, что и в Турбо Паскале — имена должны быть «видны» в том месте, где они используются, и они локализируются в пределах блока, в котором описаны.

Во встроенном ассемблере могут использоваться три предопределенных имени:

@@Code - текущий сегмент кода
 @Data - начальный сегмент данных
 @Result - ссылка внутри функции на ее результат

Имена @Code и @Data могут использоваться только в сочетании с директивой SEG для ссылки на нужный сегмент. Например:

```
asm
  mov ax, SEG @Data
  mov ds, ax
end;
```

Имя @Result используется для присвоения результата функции. Например:

```
Function Min(X,Y: Integer): Integer;
{Эта функция сравнивает два целых числа
 и возвращает наименьшее из них}
begin
  asm
    mov ax,X           {Помещаем X в AX}
    cmp ax,Y           {X<Y ?}
    jl @               {Да - на выход}
    mov ax,Y           {Нет - помещаем Y в AX}
  @: mov @Result,ax    {AX содержит результат}
  end
end;
```

Для доступа к полям записей разрешается использование составных имен. Например:

```
type
  Point = record
    X,Y: Integer
  end;
  Rect = record
    A,B: Point
  end;
var
  P: Point;
  R: Rect;
begin
  asm
    mov ax, P.X
```



```

    add ax, P.Y
    mov R.A.X, ax
end
end.

```

Идентификаторы типов можно применять к операндам для уточнения данных, о которых идет речь. Каждая из следующих команд реализует одно и то же действие: загружает в регистр AX слово по адресу ES:[DI + 4]:

```

mov ax, (Rect PTR es:[di]).B.X
mov ax, Rect(es:[di]).B.X
mov ax, es:Rect[di].B.X
mov ax, Rect[es:di].B.X
mov ax, es:[di].Rect.B.X

```

Следующие имена не могут использоваться в операндах встроенного ассемблера:

- стандартные процедуры и функции (например, *WriteLn*, *Chr*);
- предопределенные массивы *Mem*, *MemW*, *MemL*, *Port*, *PortW*;
- константы с плавающей точкой, строковые и множественного типа;
- макросы (процедуры и функции, полностью реализуемые одним *InLine* — оператором);
- символ
- *@Result* вне функции.

Выражения

Встроенный ассемблер использует выражения трех классов: регистровые, ссылки на память и непосредственные.

Регистровое выражение — это выражение, состоящее из имени регистра. Все следующие команды содержат только регистровые выражения:

```

push ds
pop  es
mov  ah,bl
add  cx,ax

```

Непосредственные выражения — это нетипизированные константы и имена типов. Примеры непосредственных выражений:

```

const
    dec = 10;
.....
asm
    mov ax, dec
    mov bx, 0
    add cx, 2*dec+1
    sub dh, -5
end;

```

Все остальные выражения относятся к ссылкам на память. Например

```

const
    dec: Word = 10;
    Step = 12;
var
    X, Y: Byte;
.....
asm
    mov ax, dec
    mov cx, [Step]
    add ah, X
    mov Y, bl
    mov ax, [bx]
end;

```

Важным отличием ассемблерных выражений от выражений Турбо Паскаля является то обстоятельство, что они должны быть статическими, т.е. разрешены (вычислены) на этапе создания программы. Если выражение может быть полностью вычислено к моменту его трансляции, т.е. если оно состоит только из регистровых или непосредственных значений, такое выражение называется абсолютным, компилятор вычисляет его и использует для создания команды.

В ходе компиляции программы вырабатывается так называемый объектный код, который затем преобразуется компоновщиком в готовую к работе программу. При создании объектного кода компилятор не может вычислить значения выражений типа «ссылка на память», так как не знает окончательного положения в памяти меток, переменных, подпрограмм. В результате он создает так называемое перемещаемое выражение, которое затем компоновщиком преобразуется в нужную ссылку на память.

Встроенный ассемблер разрешает любую операцию над абсолютным значением (см. ниже), но ограничивает перемещаемые выражения до сложения или вычитания, одним из операндов которого должна быть константа.

Другое важное отличие ассемблерных выражений от выражений Турбо Паскаля заключается в способе интерпретации переменных. В выражениях Паскаля любая ссылка на переменную интерпретируется как текущее содержимое этой переменной. В ассемблерных выражениях это справедливо только тогда, когда все выражение в целом состоит из имени переменной. Во всех остальных случаях ссылка на переменную интерпретируется как адрес переменной. Например, выражение

X+10

в Паскале означает: «к содержимому переменной X прибавить 10». В ассемблерной команде это означает: «к адресу (смещению) переменной X прибавить 10». Однако команда

mov ax, X

означает: «поместить в регистр AX первые два байта переменной X». Если бы нам понадобилось загрузить в AX адрес переменной X, мы должны были бы написать

```
mov ax,OFFSET X
```

Замечу, что попытка «перехитрить» ассемблер командами типа

```
mov ax,X+0
mov ax,X+1-1
```

и т.п. не дает желаемого результата: ассемблер просто загружает в AX содержимое переменной X.

Как и в Паскале, ассемблерные выражения имеют тип, но в отличие от Паскаля этот тип определяет только размер объекта в памяти и не ограничивает применяемые к нему операции.

Встроенный ассемблер имеет следующие предопределенные типы:

Тип	Длина в памяти
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10
NEAR	-
FAR	-

Имена предопределенных типов можно использовать для приведения типов выражений. Например, если определены переменные

```
var
  Flag: Boolean;
  X    : Word;
```

то такие ассемблерные выражения недопустимы:

```
mov Flag,bx
mov ah,X
```

Для корректного задания последней команды можно использовать следующие варианты:

```
mov ah,BYTE PTR X
mov ah,Byte(X)
mov ah,X.Byte
```

Во всех случаях в AH будет загружен первый (младший) байт переменной X.

Встроенный ассемблер поддерживает операции, перечисленные в следующей таблице (в порядке убывания приоритета).

Операции встроенного ассемблера

Операция	Комментарий
&	Перекрытие идентификатора

()	Подвыражение
[]	Ссылка на память
. (точка)	Селектор структуры
HIGH LOW	Доступ к байту в слове
+ -	Унарные операции задания знака
:	Перекрытие сегмента
OFFSET SEG TYPE PTR *	
/ MOD SHL SHR	
+ -	Бинарные операции
NOT AND OR XOR	Операции над битами

Операция &

Осуществляет перекрытие идентификатора: следующий за знаком & идентификатор считается определенным в программе, даже если он совпадает с зарезервированным словом. Например:

```
var
  Ch: Byte;
.....
mov ch,0 {Посылаем 0 в регистр CH}
mov &Ch,0 {Посылаем 0 в переменную Ch}
```

Операция ()

Круглые скобки используются обычным для Паскаля образом — для изменения порядка исчисления выражения (подвыражение, ограниченное скобками, вычисляется в первую очередь). Если перед скобками стоит имя типа, все выражение приобретает указанный тип. Например:

```
mov ax, ((1+2)*3+4)*5 {AX = 65}
mov bx, 1+2*3+4*5 {BX = 27}
```

Операция []

Определяет ссылку на память. Выражение внутри скобок вычисляется в первую очередь. Обычно оно связывается с регистрами *BX, BP, SI, DI* и может использовать операции *+* и *-* для указания индексации. Например:

```
mov ah, 100 {AH = 100}
mov ah, [100] {Загружаем в AH содержимое
байта по адресу DS:100}
```

Операция . (точка)

Селектор элемента структуры. Результат — сумма выражений до и после точки с типом второго выражения. Например:

```
var
  R: record
    X: Word; Y: Byte
  end;
.....
mov ax, R.X
mov R.Y, al
```

Операции HIGH и LOW

HIGH возвращает старший, а **LOW** — младший байт выражения типа слова, следующего за символами операции. Выражение должно иметь абсолютное непосредственное значение. Например:

```
mov al, High $1000 {AL = $10}
```

Операция : (двоеточие)

Указывает ассемблеру, что выражение после операции должно относиться к сегменту, указанному до операции. Результат — ссылка на память со значением второго выражения. Например:

```
mov ax, [10]      {AX = слово по адресу DS:10}
mov ax, BS:[10]   {AX = слово по адресу BS:10}
```

Операция OFFSET

Возвращает смещение выражения, следующего за операцией. Результат имеет непосредственное значение. Например:

```
mov ax, X          {AX = слово по адресу переменной X}
mov ax, offset X    {AX = смещение адреса X}
```

Операция SEG

Возвращает сегмент выражения, следующего за операцией. Результат имеет непосредственное значение.

Операция PTR

Осуществляет приведение типа. Результат — ссылка на память со значением выражения после операции и типом выражения до операции. Например:

```
Function Swap(X: Integer): Integer;
{Меняет местами байты в слове X}
begin
  asm
    mov ax, X
    mov BYTE PTR @Result, ah
    mov BYTE PTR @Result+1, al
  end;
end;
```

Операции * и /

* — умножение, / — целочисленное деление. Оба выражения должны иметь непосредственные абсолютные значения, такое же значение имеет и результат операции. Например:

```
mov ax, 2*2      {AX = 4}
mov ax, 17/3     {AX = 5}
```

Операция MOD

Возвращает остаток от целочисленного деления. Оба выражения должны иметь непосредственные абсолютные значения, такое же значение имеет и результат операции. Например:

```
mov ax, 17 mod 3 {AX = 2}
```

Операции SHL и SHR

Осуществляют логический сдвиг влево (SHL) или вправо (SHR) выражения, стоящего до операции, на количество разрядов, определяемое выражением после операции. Оба выражения должны иметь непосредственные абсолютные значения, такое же значение имеет и результат операции. Например:

```
mov ah, 1 shl 7 {Ah = $80 = 128}
```

Бинарная операция +

Осуществляет сложение двух выражений. Выражения могут быть непосредственными значениями или ссылками на память, но только одно из них может быть перемещаемым. Если одно из выражений — ссылка на память, результат также определяет ссылку на память, а если одно из выражений — перемещаемое, результат будет перемещаемым.

Бинарная операция -

Вычитание двух выражений. Первое выражение может быть любого класса, а второе должно быть абсолютным непосредственным значением. Результат относится к тому же классу, что и первое выражение.

Побитовые операции NOT, AND, OR, XOR

Имеют такой же смысл, что и одноименные операции Турбо Паскаля над целыми числами. Оба выражения должны иметь непосредственные абсолютные значения, такое же значение имеет и результат операции.

12.2.3. Директивы ассемблера

Встроенный ассемблер не поддерживает никакие директивы, обычно используемые в других ассемблерах, за исключением DB, DW, DD.

Структура директив такова:

```
Dx <константа> [, <константа>, ..., <константа>]
```

Здесь Dx — DB, DW или DD; <константа> — ассемблерная константа или константное выражение.

DB определяет цепочку байт, DW — слов, DD — двойных слов. Например:

```
db 'Турбо Паскаль', 13, 10
dw 0, $FFFF, NearProc
dd 'ABCD', 999999999, FarProc
```

В качестве константных выражений разрешается использовать любые ассемблерные константы со значением, не выходящим из диапазона байта (*DB*), слова (*DW*) или двойного слова (*DD*). В любой директиве можно опеределять строковую константу, которая приводит к побайтовому заполнению памяти *ASCII*-кодами символов. Поскольку слово (двойное слово) размещается в памяти, начиная со своего младшего байта, старший (старшие) байт в директивах *DW* и *DD* при размещении строкой константы может остаться неопределенным и заполняется нулем. Например, два следующих объявления эквивалентны:

```
dw      '5'
dw      $35 { $35 - ASCII-код символа '5' }
```

В директивах *DW* и *DD* разрешается также указывать имена, которые в этом случае интерпретируются как адреса соответствующих объектов, причем для *DW* это — ближний адрес (смещение), а для *DD* — дальний. Например:

```
dw X      {Размещает смещение переменной X}
dd Proc   {Размещает FAR-адрес процедуры Proc}
```

Данные, определяемые директивами *Dx*, всегда размещаются в текущем кодовом сегменте. Разместить таким образом данные в сегменте данных (т.е. определить константу или типизированную константу) невозможно — для этого используются стандартные средства Турбо Паскаля. Более того, директивы не могут снабжаться именами, а поэтому использовать размещаемые с их помощью данные не так — то просто. В следующем примере на экран выводится текстовое сообщение. Для этого используется функция 9 вызова ДОС, в соответствии с которой в регистрах *DS:DX* должен содержаться адрес текстовой строки, а сама строка должна заканчиваться символом «\$»:

```
asm
  jmp @NextCode      {Обходим фрагмент данных}
@:
  db 'Текстовая строка,13,10','$'
@NextCode:
  push ds             {Сохраняем DS}
  push cs
  pop ds              {DS = CS}
  mov dx,OFFSET @     {DS:DX - адрес строки}
  mov ah,9            {AH - код функции вывода}
  int 21h             {Выводим строку}
  pop ds              {Восстанавливаем DS}
end;
```

Обратите внимание на использование регистра *DS*. В соответствии с требованиями функции 9, он должен содержать сегмент выводимой строки. В нашем случае строка располагается в кодовом сегменте, поэтому мы вынуждены сначала сохранить значение *DS* в стеке, а затем восстановить его. Если бы мы этого не сделали, по завершении ассемблерного оператора

регистр *DS* указывал бы на сегмент кода и была бы потеряна связь программы Турбо Паскаля с глобальными переменными и константами.

12.2.4. Ассемблерные подпрограммы

Ассемблерные подпрограммы — это процедуры и функции, объявленные с директивой **Assembler**. В таких подпрограммах исполняемая часть не содержит **begin... end** и состоит из единственного ассемблерного оператора **asm... end**. Например:

```
Function LongMul (X,Y:Integer):LongInt; Assembler;
asm
    mov  ax, X
    imul Y    {DX/AX содержат "длинный" результат}
end;
```

При компиляции ассемблерных подпрограмм выполняется ряд оптимизаций кода, в том числе:

- параметры — значения строкового типа, а также длиной в 1, 2 и 4 байта не копируются во временную память, т.е. внутри подпрограммы они считаются параметрами — переменными;
- компилятор не создает переменную *@Result* для результата функции, и ссылка на эту переменную в ассемблерной функции недопустима; исключением являются функции, возвращающие значения строкового типа, для них разрешается использовать ссылку на *@Result*;
- генерируются следующие команды на входе в подпрограмму и на ее выходе:

```
push bp          {Сохраняется BP}
mov  bp,sp       {BP содержит текущую границу стека}
sub  sp,Locals   {Резервируется часть стека для
                  размещения локальных переменных}
.....
mov  sp,bp       {Восстанавливается граница стека}
pop  bp          {Восстанавливается BP}
ret  Params      {Из стека удаляются параметры
                  подпрограммы и осуществляется выход из нее}
```

Здесь *Locals* — общая длина в байтах всех объявленных в подпрограмме локальных переменных, а *Params* — длина (в байтах) всех формальных параметров. Если *Locals* и *Params* равны нулю, входной код не создается, а выходной содержит единственную инструкцию **RET**.

Все локальные переменные Турбо Паскаль размещает в стеке. Это относится как к обычным, так и к ассемблерным подпрограммам. Для ссылки на локальные переменные используется адресация по базе, задаваемой парой *DS: BP*, поэтому при входе в процедуру всегда создается так называемый локальный стек: в регистр *BP* помещается текущая граница стека, а сама эта граница смещается вверх на суммарную длину всех локальных переменных, чтобы работа со стеком внутри подпрограммы не разрушила локальные переменные. Например:


```
Procedure ....; Assembler;  
var  
  X: Word;  
  Y: Byte;  
asm  
  mov X, ax {Компилируется в mov [BP-2], ax}  
  mov ah, Y {Компилируется в mov ah, [BP-3]}  
end;
```

Ассемблерные функции должны следующим образом возвращать результат своей работы:

- длиной 1 байт (*Byte*, *Char* и т.п.) — в регистре *AL*;
- длиной 2 байта (*Integer*, *Word*) — в регистре *AX*;
- длиной 4 байта (*Pointer*, *LongInt*) — в регистрах *DX* (старшее слово) и *AX* (младшее слово);
- типа *Real* — в регистрах *DX*, *BX*, *AX* (старшее слово — в *DX*, младшее в *AX*);
- вещественных типов *Single*, *Double*, *Extended*, *Comp* — в регистре *ST* (0) сопроцессора;
- строкового типа — во временной области памяти, на которую ссылается *@Result*.

ИСПОЛЬЗОВАНИЕ БИБЛИОТЕКИ CRT

Во многих случаях стандартные для Паскаля возможности ввода/вывода данных с помощью процедур *Read*, *ReadLn*, *Write*, *WriteLn* оказываются явно недостаточными для разработки удобных в использовании диалоговых программ. Например, процедуры *Read/ReadLn* вводят с клавиатуры только типизированные данные, причем с обязательным эхо-повтором набранных символов на экране. С их помощью нельзя определить факт нажатия какой-либо специальной клавиши (функциональной клавиши, клавиши управления курсором и т.п.). Процедуры *Write/WriteLn* выводят сообщения, начиная с того места на экране, где в данный момент находится курсор, причем по мере вывода курсор автоматически сдвигается на экране, а если очередной символ выводится в самом нижнем правом углу экрана, осуществляется «прокрутка» экрана: его содержимое сдвигается вверх на одну строку. Все это сильно затрудняет создание и обновление различного рода окон, меню и других атрибутов современных диалоговых программ.

Разработчики Турбо Паскаля предусмотрели несколько подпрограмм, существенно увеличивающих возможности текстового ввода/вывода. Эти подпрограммы сосредоточены в библиотеке (модуле) *CRT*, входящей в комплект поставки Турбо Паскаля. В модуль включены также процедуры *Sound*, *NoSound* и *Delay*, которые позволяют программировать звуковой генератор ПК. В этой главе обсуждается использование подпрограмм модуля *CRT*¹.

13.1. ПРОГРАММИРОВАНИЕ КЛАВИАТУРЫ

Дополнительные возможности управления клавиатурой реализуются двумя функциями: *KeyPressed* и *ReadKey*.

Функция *KeyPressed*. Возвращает значение типа *Boolean*, указывающее состояние буфера клавиатуры: *False* означает, что буфер пуст, а *True* — что в буфере есть хотя бы один символ, еще не прочитанный программой.

В *MS-DOS* реализуется так называемый асинхронный буферизованный ввод с клавиатуры. По мере нажатия на клавиши соответствующие коды помещаются в особый буфер, откуда они могут быть затем прочитаны программой. Стандартная длина буфера рассчитана на хранение до 16 кодов символов. Если программа достаточно долго не обращается к клавиатуре, а пользователь нажимает клавиши, буфер может оказаться переполненным. В этот момент раздается звуковой сигнал и «лишние» коды

¹ Аббревиатура *CRT* соответствует русскоязычной аббревиатуре ЭЛТ — электронная лучевая трубка. На профессиональном жаргоне *CRT* означает устройство визуализации информации (дисплей) даже в том случае, когда вместо ЭЛТ используются иные физические устройства — плазменные панели, жидкокристаллические экраны и т.п.

теряются. Чтение из буфера обеспечивается процедурами *Read/ReadLn* и функцией *ReadKey*. Замечу, что обращение к функции *KeyPressed* не задерживает исполнения программы: функция немедленно анализирует буфер и возвращает то или иное значение, не дожидаясь нажатия клавиши.

Функция *ReadKey*. Возвращает значение типа *Char*. При обращении к этой функции анализируется буфер клавиатуры: если в нем есть хотя бы один не прочитанный символ, код этого символа берется из буфера и возвращается в качестве значения функции, в противном случае функция будет ожидать нажатия на любую клавишу. Ввод символа с помощью этой функции не сопровождается эхо-повтором и содержимое экрана не меняется.

Пусть, например, в какой-то точке программы необходимо игнорировать все ранее нажатые клавиши, коды которых еще не прочитаны из буфера, т.е. необходимо очистить буфер. Этого можно достичь следующим способом:

```
Uses CRT;  
var  
  C: Char;  
begin  
  while KeyPressed do  
    C := ReadKey;  
    .....  
end.
```

При использовании процедуры *ReadKey* необходимо учесть, что в клавиатурный буфер помещаются так называемые расширенные коды нажатых клавиш. Если нажимается любая алфавитно-цифровая клавиша, расширенный код совпадает с *ASCII*-кодом соответствующего символа. Например, если нажимается клавиша с латинской буквой «a» (в нижнем регистре), функция *ReadKey* возвращает значение *chr(97)*, а если «A» (в верхнем регистре) — значение *chr(65)*. При нажатии функциональных клавиш *F1...F10*, клавиш управления курсором, клавиш *Ins*, *Home*, *Del*, *End*, *PgUp*, *PgDn* в буфер помещается двухбайтная последовательность: сначала символ #0, а затем расширенный код клавиши. Таким образом, значение #0, возвращаемое функцией *ReadKey*, используется исключительно для того, чтобы указать программе на генерацию расширенного кода. Получив это значение, программа должна еще раз обратиться к функции, чтобы прочитать расширенный код клавиши¹.

Следующая простая программа позволит Вам определить расширенный код любой клавиши. Для завершения работы программы нажмите клавишу *Esc*.

```
Uses CRT;
```

¹ Т.е. код сканирования клавиши. Этот код определяется порядком, в соответствии с которым микропроцессор клавиатуры *Intel 8042* периодически опрашивает (сканирует) состояние клавиш.

```

var
  C: Char;
begin
  repeat
    C := ReadKey;
    if C<>#0 then
      WriteLn(ord(C))
    else
      WriteLn('0',ord(ReadKey):8)
  until C=#27 {27 - расширенный код клавиши Esc}
end.

```

Если Вы воспользуетесь этой программой, то обнаружите, что нажатие на некоторые клавиши игнорируется функцией *ReadKey*. Это прежде всего так называемые сдвиговые клавиши — *Shift*, *Ctrl*, *Alt*. Сдвиговые клавиши в *MS-DOS* обычно используются для переключения регистров клавиатуры и нажимаются в сочетании с другими клавишами. Именно таким способом, например, различается ввод прописных и строчных букв. Кроме того, функция игнорирует переключающие клавиши *Caps Lock*, *Num Lock*, *Scroll Lock*, а также «лишние» функциональные клавиши *F11* и *F12* клавиатуры *IBM AT*, не имеющие аналога на клавиатуре ранних моделей *IBM PC/XT* (в этих машинах использовалась 84-клавишная клавиатура, в то время как на *IBM AT* — 101-клавишная).

В табл.13.1 приводятся расширенные коды клавиш, возвращаемые функцией *ord(ReadKey)*. Для режима ввода кириллицы приводятся коды, соответствующие альтернативному варианту кодировки.

Таблица 13.1

Расширенные коды клавиш

Код		Клавиша или комбинация клавиш	Код		Клавиша или комбинация клавиш
Первый байт	Второй байт		Первый байт	Второй байт	
Алфавитно–цифровые клавиши					
8	—	Backspace (Забой)	9	—	Tab (Табуляция)
13	—	Enter	32	—	Пробел
33	—	!	34	—	"
35	—	#	36	—	\$
37	—	%	38	—	&
39	—	'	40	—	(
41	—	)	42	—	*
43	—	+	44	—	,
45	—	—	46	—	.
47	—	/	48...57	—	0...9
58	—	:	59	—	;
60	—	<	61	—	=
62	—	>	63	—	?

64	—	@	65...90	—	A...Z
91	—	[	92	—	\
93	—	]	94	—	
95	—	=	96	—	.
97...122	—	a...z	123	—	{
124	—		125	—	}
126	—		128...159	—	A...Я
160...175	—	a...п	224...239	—	р...я
Управляющие клавиши и их сочетания со сдвиговыми					
0	3	Ctrl—2	0	15	Shift—Tab
0	16...25	Alt—Q...Alt—P (верхний ряд букв)	0	30...38	Alt—A...Alt—L (средний ряд букв)
0	44...50	Alt—Z...Alt—M (нижний ряд букв)	0	59...68	F1...F10
0	71	Home	0	72	Курсор вверх
0	73	PgUp	0	75	Курсор влево
0	77	Курсор вправо	0	79	End
0	80	Курсор вниз	0	81	PgDn
0	82	Ins	0	83	Del
0	84...93	Shift—F1...Shift—F10	0	94...103	Ctrl—F1... Ctrl—F10
0	104...113	Alt—F1...Alt—F10	0	114	Ctrl—PrtScr
0	115	Ctrl—курсор влево	0	116	Ctrl—Курсор вправо
0	117	Ctrl—End	0	118	Ctrl—PgDn
0	119	Ctrl—Home	0	120...131	Alt—1...Alt—=
0	132	Ctrl—PgUp			(верхний ряд кла — виш)

13.2. ТЕКСТОВЫЙ ВЫВОД НА ЭКРАН

Библиотека Turbo Vision способна удовлетворить самым высоким требованиям и я настоятельно рекомендую обращаться к ней при программировании сложных текстовых изображений (меню, окон и т.п.). Тем не менее вполне возможно, что некоторые из читателей захотят использовать значительно более простые, но достаточно эффективные средства модуля CRT, описываемые в этом разделе.

Используемое в ПК устройство визуального отображения информации — дисплей — состоит из двух основных частей: монитора, содержащего экран (электронно — лучевую трубку или жидкокристаллическую панель) с необходимыми компонентами (устройствами развертки изображения), и блока управления, который чаще называют дисплейным адаптером или просто адаптером. Обычно оба устройства согласуются друг с другом, но в отдельных случаях этого согласования может не быть (например, цветной монитор может работать с монохромным адаптером и наоборот). Будем считать оба устройства согласованными, поэтому, говоря о различных дисплеях, я буду говорить только о различных адаптерах, так как именно в них сосредоточены основные отличия дисплеев друг от друга.

Исторически первым адаптером (1981 г.), использованным на IBM PC, был так называемый монохромный адаптер (MDA). Его возможности очень скромны: он позволял выводить только текстовые сообщения в одном из

двух форматов — 25 строк по 40 или по 80 символов в строке. Символы выводились в прямом изображении (светлые символы на темном фоне), причем их ширина оставалась одинаковой в обоих режимах, поэтому при выводе в режиме 40x25 использовалась только левая половина экрана. В MDA применялись два символьных шрифта — обычный и с подчеркиванием.

В 1982 году фирма *Hercules* выпустила адаптер HGC (от англ. *Hercules Graphics Card* — графическая карта Геркулес), который полностью эмулировал MDA в текстовом режиме, но в отличие от него мог еще воспроизводить и графические изображения с разрешением 720x350 точек (пиксел).

Примерно в это же время IBM выпустила цветной графический адаптер CGA (*Color Graphics Adapter*) и впервые на экране ПК появился цвет. CGA позволяла выводить как текстовые сообщения, так и графические изображения (с разрешением 320x200 или 640x200 пиксел). В текстовом режиме выводились 40x25 или 80x25 символов как в монохромном, так и в цветном изображениях. При использовании монохромного режима символы, в отличие от MDA, не могли подчеркиваться, зато их можно было выводить в негативном изображении (черные символы на светлом фоне). При выводе в цветном режиме использовалось 16 цветов для символов и 8 — для окружающего их фона.

Текстовые возможности CGA стали стандартом де-факто и поддерживаются во всех последующих разработках IBM — адаптерах EGA, MCGA, VGA и SVGA. Возможности модуля CRT рассматриваются применительно к адаптерам этого типа.

Процедура TextMode. Используется для задания одного из возможных текстовых режимов работы адаптера. Заголовок процедуры:

Procedure TextMode (Mode: Word);

Здесь *Mode* — код текстового режима. В качестве значения этого выражения могут использоваться следующие константы, определенные в модуле CRT:

const

```

BW40    = 0;    {Черно-белый режим 40x25}
Co40    = 1;    {Цветной режим 40x25}
BW80    = 2;    {Черно-белый режим 80x25}
Co80    = 3;    {Цветной режим 80x25}
Mono    = 7;    {Используется с MDA}
Font8x8 = 256;  {Используется для загружаемого шрифта в
                  режиме 80x43 или 80x50 с адаптерами EGA или VGA}

```

Код режима, установленного с помощью вызова процедуры *TextMode*, запоминается в глобальной переменной *LastMode* модуля CRT и может использоваться для восстановления начального состояния экрана.

Следующая программа иллюстрирует использование этой процедуры в различных режимах. Замечу, что при вызове *TextMode* сбрасываются все

ранее сделанные установки цвета и окон, экран очищается и курсор переводится в его левый верхний угол.

Uses CRT;

Procedure Print(S: String);

{Выводит сообщение S и ждет инициативы пользователя}

begin

WriteLn(S); {Выводим сообщение}

WriteLn('Нажмите клавишу Enter...');

ReadLn {Ждем нажатия клавиши Enter}

end; {Print}

var

LM: Word; {Начальный режим экрана}

begin

LM := LastMode; {Запоминаем начальный режим работы дисплея}

TextMode(Co40);

Print('Режим 40x25');

TextMode(Co80);

Print('Режим 80x25');

TextMode(Co40+Font8x8);

Print('Режим Co40+Font8x8');

TextMode(Co80+Font8x8);

Print('Режим Co80+Font8x8');

{Восстанавливаем исходный режим работы:}

TextMode(LM)

end.

Процедура TextColor. Определяет цвет выводимых символов. Заголовок процедуры:

Procedure TextColor(Color: Byte);

Процедура TextBackground. Определяет цвет фона. Заголовок:

Procedure TextBackground(Color: Byte);

Единственным параметром обращения к этим процедурам должно быть выражение типа *Byte*, задающее код нужного цвета. Этот код удобно определять с помощью следующих мнемонических констант, объявленных в модуле CRT:

const

Black	=	0; {Черный}
Blue	=	1; {Темно-синий}
Green	=	2; {Темно-зеленый}
Cyan	=	3; {Бирюзовый}
Red	=	4; {Красный}
Magenta	=	5; {Фиолетовый}
Brown	=	6; {Коричневый}

```

LightGray      = 7; {Светло-серый}
DarkGray       = 8; {Темно-серый}
LightBlue      = 9; {Синий}
LightGreen     = 10; {Светло-зеленый}
LightCyan      = 11; {Светло-бирюзовый}
LightRed       = 12; {Розовый}
LightMagenta   = 13; {Малиновый}
Yellow         = 14; {Желтый}
White          = 15; {Белый}
Blink          =128; {Мерцание символа}

```

Следующая программа иллюстрирует цветовые возможности Турбо Паскаля.

```

Uses CRT;
const
  Col: array [1..15] of String [16] =
    ('темно-синий', 'темно-зеленый', 'бирюзовый', 'красный',
     'фиолетовый', 'коричневый', 'светло-серый', 'темно-серый',
     'синий', 'зеленый', 'светло-бирюзовый', 'розовый',
     'малиновый', 'желтый', 'белый');
var
  k: Byte;
begin
  for k := 1 to 15 do
    begin           {Выводим 15 сообщений различными цветами}
      TextColor(k);
      WriteLn('Цвет ', k, ' - ', Col[k])
    end;
  TextColor(White+Blink); {Белые мигающие символы}
  WriteLn('Мерцание символов');
  {Восстанавливаем стандартный цвет}
  TextColor(LightGray);
  WriteLn
end.

```

Обратите внимание на последний оператор *WriteLn*: если его убрать, режим мерцания символов сохранится после завершения программы, несмотря на то, что перед ним стоит оператор

```
TextColor(LightGray)
```

Дело в том, что все цветовые определения предварительно заносятся в специальную переменную *TextAttr* модуля *CRT* и используются для настройки адаптера только при обращении к процедурам *Write/WriteLn*.

Процедура *ClrScr*. Очищает экран или окно (см. ниже процедуру *Window*). После обращения к ней экран (окно) заполняется цветом фона и курсор устанавливается в его левый верхний угол. Например:

```
Uses CRT;
```



```
var
  C: Char
begin
  TextBackground(red);
  ClrScr;                                {Заполняем экран красным цветом}
  WriteLn('Нажмите любую клавишу...');
  C := ReadKey;                          {Ждем нажатия любой клавиши}
  TextBackground(Black);
  ClrScr                                {Восстанавливаем черный фон экрана}
end.
```

Процедура Window. Определяет текстовое окно — область экрана, которая в дальнейшем будет рассматриваться процедурами вывода как весь экран. Сразу после вызова процедуры курсор помещается в левый верхний угол окна, а само окно очищается (заполняется цветом фона). По мере вывода курсор, как обычно, смещается вправо и при достижении правой границы окна переходит на новую строку, а если он к этому моменту находился на последней строке, содержимое окна сдвигается вверх на одну строку, т.е. осуществляется «прокрутка» окна. Заголовок процедуры:

Procedure Window(X1,Y1,X2,Y2: Byte);

Здесь $X1...Y2$ — координаты левого верхнего ($X1,Y1$) и правого нижнего ($X2,Y2$) углов окна. Они задаются в координатах экрана, причем левый верхний угол экрана имеет координаты (1,1), горизонтальная координата увеличивается слева направо, а вертикальная — сверху вниз.

В следующем примере иллюстрируется вывод достаточно длинного сообщения в двух разных окнах.

```
Uses CRT;
var
  k: integer;
begin
  {Создаем левое окно -желтые символы на синем фоне:}
  TextBackground(Blue);
  Window(5,2,35,17);
  TextColor(Yellow);
  for k := 1 to 100 do
    Write(' Нажмите клавишу Enter...');
  ReadLn;                                {Ждем нажатия Enter}
  ClrScr;                                {Очищаем окно}
  {Создаем правое окно - белые символы на красном фоне:}
  TextBackground(Red);
  TextColor(White);
  Window(40,2,70,17);
  for k := 1 to 100 do
    Write(' Нажмите клавишу Enter...');
  ReadLn;
```

```
TextMode(C080)           {Сбрасываем все установки}
end.
```

Обращение к процедуре *Window* игнорируется, если какая-либо из координат выходит за границы экрана или если нарушается одно из условий: $X2 > X1$ и $Y2 > Y1$. Каждое новое обращение к *Window* отменяет предыдущее определение окна. Границы текущего окна запоминаются в двух глобальных переменных модуля *CRT*: переменная *WindMin* типа *Word* хранит $X1$ и $Y1$ ($X1$ — в младшем байте), а переменная того же типа *WindMax* — $X2$ и $Y2$ ($X2$ — в младшем байте). При желании Вы можете изменять их нужным образом без обращения к *Window*. Например, вместо оператора

```
Window(40,2,70,17);
```

можно было бы использовать два оператора

```
WindMin := 39+(1 shl 8);
WindMax := 69+(16 shl 8);
```

(в отличие от обращения к *Window* координаты, хранящиеся в переменных *WindMin* и *WindMax*, соответствуют началу отсчета 0,0).

Процедура GotoXY. Переводит курсор в нужное место экрана или текущего окна. Заголовок процедуры:

```
Procedure GotoXY(X,Y: Byte);
```

Здесь X, Y — новые координаты курсора. Координаты задаются относительно границ экрана (окна), т.е. оператор

```
GotoXY(1,1);
```

означает указание перевести курсор в левый верхний угол экрана (или окна, если к этому моменту на экране определено окно). Обращение к процедуре игнорируется, если новые координаты выходят за границы экрана (окна).

Функции WhereX и WhereY. С помощью этих функций типа *Byte* можно определить текущие координаты курсора: *WhereX* возвращает его горизонтальную, а *WhereY* — вертикальную координаты.

В следующей программе сначала в центре экрана создается окно, которое обводится рамкой, затем в окне выводится таблица из двух колонок.

```
Uses CRT;
```

```
const
```

```
LU = #218; {Левый верхний угол рамки}
RU = #191; {Правый верхний угол}
LD = #192; {Левый нижний}
RD = #217; {Правый нижний}
H  = #196; {Горизонтальная черта}
V  = #179; {Вертикальная черта}
X1 = 14;   {Координаты окна}
```

```

Y1 = 5;
X2 = 66;
Y2 = 20;
Txt = 'Нажмите клавишу Enter...';
var
  k: integer;
begin
  ClrScr;                                {Очищаем экран}
  {Создаем окно в центре экрана - желтые символы на синем фоне:}
  TextBackground(Blue);
  TextColor(Yellow);
  Window(X1,Y1,X2,Y2);
  ClrScr;
  {Обводим окно рамкой}
  Write(LU);                             {Левый верхний угол}
  {Горизонтальная линия}
  for k:= X1+1 to X2-1 do Write(H);
  Write(RU);                             {Верхний правый угол}
  for k := Y1+1 to Y2-1 do               {Вертикальные линии}
    begin
      GotoXY(1,k-Y1+1);                  {Переходим к левой границе}
      Write(V);                          {Левая черта}
      GotoXY(X2-X1+1,WhereY); {Правая граница}
      Write(V)                          {Правая черта}
    end;
  Write(LD);                             {Левый нижний угол}
  Window(X1,Y1,X2,Y2+1);                 {Расширяем вниз на одну
                                         строку координаты окна, иначе вывод в правый
                                         нижний угол вызовет прокрутку окна вверх}
  GotoXY(2,Y2-Y1+1);                    {Возвращаем курсор из левого
                                         верхнего угла окна на нужное место}
  {Горизонтальная рамка}
  for k:= X1+1 to X2-1 do Write(H);
  Write(RD);                             {Правый нижний угол}
  {Определяем внутреннюю часть окна}
  Window(X1+1,Y1+1,X2-1,Y2-1);
  {Выводим левый столбец}
  for k := Y1+1 to Y2-2 do
    WriteLn('Левый столбец, строка ',k-Y1);
  {Ждем нажатия любой клавиши}
  Write('Нажмите любую клавишу...');
  k := ord(ReadKey);
  if k=0 then
    k := ord(ReadKey);
  DeleteLine;                            {Стираем приглашение}
  {Выводим правый столбец}
  for k := Y1+1 to Y2-2 do
    begin

```

```

    GotoXY((X2-X1) div 2,k-Y1);
    Write('Правый столбец, строка ',k-Y1)
end;
{Выводим сообщение и ждем нажатия клавиши Enter}
GotoXY((X2-X1-Length(Txt)) div 2,Y2-Y1-1);
TextColor(White);
Write(Txt);
ReadLn;
{Восстанавливаем стандартный режим}
TextMode(CO80)
end.

```

Три следующие процедуры без параметров могут оказаться полезными при разработке текстовых редакторов.

Процедура ClrEOL. Стирает часть строки от текущего положения курсора до правой границы окна (экрана). Положение курсора не меняет — ся.

Процедура DelLine. Уничтожает всю строку с курсором в текущем окне (или на экране, если окно не создано). При этом все строки ниже удаляемой (если они есть) сдвигаются вверх на одну строку.

Процедура InsLine. Вставляет строку: строка с курсором и все строки ниже ее сдвигаются вниз на одну строку; строка, вышедшая за нижнюю границу окна (экрана), безвозвратно теряется; текущее положение курсора не меняется.

Процедуры LowVideo, NormVideo и HighVideo. С помощью этих процедур без параметров можно устанавливать соответственно пониженную, нормальную и повышенную яркость символов. Например:

```

Uses CRT;
begin
    LowVideo;
    WriteLn('Пониженная яркость');
    NormVideo;
    WriteLn('Нормальная яркость');
    HighVideo;
    WriteLn('Повышенная яркость')
end.

```

Замечу, что на практике нет разницы между пониженной и нормальной яркостью изображения.

Процедура AssignCRT. Связывает текстовую файловую переменную *F* с экраном с помощью непосредственного обращения к видеопамати (т.е. к памяти, используемой адаптером для создания изображения на экране). В результате вывод в такой текстовый файл осуществляется значительно (в 3...5 раз) быстрее, чем если бы этот файл был связан с экраном стандартной процедурой *Assign*. Заголовок процедуры:

Procedure AssignCRT(F: Text);

В следующей программе измеряется скорость вывода на экран с помощью стандартной файловой процедуры и с помощью непосредственного обращения к видеопамати. Вначале файловая переменная F связывается «медленной» процедурой *Assign* со стандартным устройством CON (т.е. с экраном) и подсчитывается количество N1 циклов вывода некоторого текста за $5 \cdot 55 = 275$ миллисекунд системных часов. Затем файловая переменная связывается с экраном с помощью процедуры быстрого доступа *AssignCRT* и точно так же подсчитывается количество N2 циклов вывода. В конце программы счетчики N1 и N2 выводятся на экран.

Замечу, что показания системных часов хранятся в оперативной памяти компьютера в виде четырехбайтного слова по адресу [\$0040:\$006C] и наращиваются на единицу каждые 55 миллисекунд.

Uses CRT;

var

F: Text;
t: LongInt; {Начало отсчета времени}
N1, N2: Word; {Счетчики вывода}

const

txt = ' Text';

begin

{----- Стандартный вывод в файл -----}

Assign(F, 'CON');

Rewrite(F);

N1 := 0; {Готовим счетчик вывода}

ClrScr; {Очищаем экран}

{Запоминаем начальный момент:}

t := MemL[\$0040:\$006C];

{Ждем начала нового 55-мс интервала, чтобы
исключить погрешность в определении времени:}

while MemL[\$0040:\$006C]=t **do**;

{Цикл вывода за 5 интервалов}

while MemL[\$0040:\$006C]<t+6 **do**

begin

inc(N1);

Write(F, txt)

end;

Close(F);

{----- Вывод с помощью быстрой процедуры
прямого доступа к экрану -----}

AssignCRT(F);

Rewrite(F);

N2 := 0;

ClrScr;

t := MemL[\$0040:\$006C];

while MemL[\$0040:\$006C]=t **do**;

while MemL[\$0040:\$006C]<t+6 **do**

```

begin
    inc(N2);
    Write(F,txt)
end;
Close(F);
{Печатаем результат}
ClrScr;
WriteLn(N1,N2:10)
end.

```

Следует учесть, что вывод на экран обычным образом — без использования файловой переменной (например, оператором `Write(txt)`) также осуществляется с помощью непосредственного доступа к видеопамати, поэтому ценность процедуры *AssignCRT* весьма сомнительна. Прямой доступ к видеопамати регулируется глобальной логической переменной *DirectVideo* модуля *CRT*: если эта переменная имеет значение *True*, доступ разрешен, если *False* — доступ к экрану осуществляется с помощью относительно медленных средств операционной системы *MS-DOS*. По умолчанию переменная *DirectVideo* имеет значение *True*.

13.3. ПРОГРАММИРОВАНИЕ ЗВУКОВОГО ГЕНЕРАТОРА

Звуковые возможности ПК основаны на одноканальном управляемом звуковом генераторе, вырабатывающем электромагнитные колебания звуковой частоты. Колебания подаются на встроенный в ПК динамик и заставляют его звучать.

В модуль *CRT* включены три процедуры, с помощью которых Вы сможете запрограммировать произвольную последовательность звуков.

Процедура *Sound*. Заставляет динамик звучать с нужной частотой. Заголовок процедуры:

Procedure Sound(F: Word);

Здесь *F* — выражение типа *Word*, определяющее частоту звука в герцах. После обращения к процедуре включается динамик и управление немедленно возвращается в основную программу, в то время как динамик будет звучать впредь до вызова процедуры *NoSound*.

Процедура *NoSound*. Выключает динамик. Если он к этому моменту не был включен, вызов процедуры игнорируется.

Процедура *Delay*. Обеспечивает задержку работы программы на заданный интервал времени. Заголовок процедуры:

Procedure Delay(T: Word);

Здесь *T* — выражение типа *Word*, определяющее интервал времени (в миллисекундах), в течение которого задерживается выполнение следующего оператора программы.

Для генерации звукового сигнала обычно используется вызов описанных процедур по схеме *Sound-Delay-NoSound*. Следующая программа

заставит ПК воспроизвести простую музыкальную гамму. Используемый в ней массив *F* содержит частоты всех полутонов в первой октаве от «до» до «си». При переходе от одной октавы к соседней частоты изменяются в два раза.

Uses CRT;

const

```
F: array [1..12] of Real =  
  (130.8, 138.6, 146.8, 155.6, 164.8,  
   174.6, 185.0, 196.0, 207.7, 220.0,  
   233.1, 246.9);           {Массив частот 1-й октавы}  
Temp = 100;                 {Темп исполнения}
```

var

k,n: Integer;

begin

{Восходящая гамма}

for k := 0 **to** 3 **do**

for n := 1 **to** 12 **do**

begin

 Sound(Round(F[n]*(1 shl k)));

 Delay(Temp);

 NoSound

end;

{Нисходящая гамма}

for k := 3 **downto** 0 **do**

for n := 12 **downto** 1 **do**

begin

 Sound(Round(F[n]*(1 shl k)));

 Delay(Temp);

 NoSound

end

end.

ИСПОЛЬЗОВАНИЕ БИБЛИОТЕКИ GRAPH

Начиная с версии 4.0, в состав Турбо Паскаля включена мощная библиотека графических подпрограмм *Graph*, остающаяся практически неизменной во всех последующих версиях. Библиотека содержит в общей сложности более 50 процедур и функций, предоставляющих программисту самые разнообразные возможности управления графическим экраном. Для облегчения знакомства с библиотекой все входящие в нее процедуры и функции сгруппированы по функциональному принципу.

14.1. ПЕРЕХОД В ГРАФИЧЕСКИЙ РЕЖИМ И ВОЗВРАТ В ТЕКСТОВЫЙ

Стандартное состояние ПК после его включения, а также к моменту запуска программы из среды Турбо Паскаля соответствует работе экрана в текстовом режиме, поэтому любая программа, использующая графические средства компьютера, должна определенным образом инициализировать графический режим работы дисплейного адаптера. После завершения работы программы ПК возвращается в текстовый режим.

14.1.1. Краткая характеристика графических режимов работы дисплейных адаптеров

Настройка графических процедур на работу с конкретным адаптером достигается за счет подключения нужного графического драйвера. Драйвер — это специальная программа, осуществляющая управление теми или иными техническими средствами ПК. Графический драйвер, как это не трудно догадаться, управляет дисплейным адаптером в графическом режиме. Графические драйверы разработаны фирмой *Borland* практически для всех типов адаптеров. Обычно они располагаются на диске в отдельном подкаталоге *BGI* в виде файлов с расширением *BGI* (от англ.: *Borland Graphics Interface* — графический интерфейс фирмы *Borland*). Например, *CGA.BGI* — драйвер для *CGA*-адаптера, *EGAVGA.BGI* — драйвер для адаптеров *EGA* и *VGA* и т.п.

Выпускаемые в настоящее время ПК оснащаются адаптерами, разработанными фирмой *IBM*, или совместимыми с ними. Если не учитывать уже упоминавшийся в гл.13 монохромный адаптер *MDA*, все они имеют возможность работы в графическом режиме. В этом режиме экран дисплея рассматривается как совокупность очень близко расположенных точек — пиксел, светимостью которых можно управлять с помощью программы.

Графические возможности конкретного адаптера определяются разрешением экрана, т.е. общим количеством пиксел, а также количеством цветов (оттенков), которыми может светиться любой из них. Кроме того, многие адаптеры могут работать с несколькими графическими страницами. Графической страницей называют область оперативной памяти, исполь-

зубая для создания «карты» экрана, т.е. содержащая информацию о светимости (цвете) каждого пиксела. Ниже приводится краткая характеристика графических режимов работы наиболее распространенных адаптеров.

Адаптер *CGA* (*Color Graphics Adapter* — цветной графический адаптер) имеет 5 графических режимов. Четыре режима соответствуют низкой разрешающей способности экрана (320 пиксел по горизонтали и 200 по вертикали, т.е. 320x200) и отличаются только набором допустимых цветов — палитрой. Каждая палитра состоит из трех цветов, а с учетом черного цвета несветящегося пиксела — из четырех: палитра 0 (светло-зеленый, розовый, желтый), палитра 1 (светло-бирюзовый, малиновый, белый), палитра 2 (зеленый, красный, коричневый) и палитра 3 (бирюзовый, фиолетовый, светло-серый). Пятый режим соответствует высокому разрешению 640x200, но каждый пиксел в этом случае может светиться либо каким-то одним заранее выбранным и одинаковым для всех пикселов цветом, либо не светиться вовсе, т.е. палитра этого режима содержит два цвета. В графическом режиме адаптер *CGA* использует только одну страницу.

Адаптер *EGA* (*Enhanced Graphics Adapter* — усиленный графический адаптер) может полностью эмулировать графические режимы адаптера *CGA*. Кроме того, в нем возможны режимы: низкого разрешения (640x200, 16 цветов, 4 страницы) и высокого разрешения (640x350, 16 цветов, 1 страница). В некоторых модификациях используется также монохромный режим (640x350, 1 страница, 2 цвета).

Адаптер *MCGA* (*Multi-Color Graphics Adapter* — многоцветный графический адаптер) совместим с *CGA* и имеет еще один режим — 640x480, 2 цвета, 1 страница. Такими адаптерами оснащались младшие модели серии ПК *PS/2* фирмы *IBM*. Старшие модели этой серии оснащаются более совершенными адаптерами *VGA* (*Video Graphics Array* — графический видеомассив). Адаптер *VGA* эмулирует режимы адаптеров *CGA* и *EGA* и дополняет их режимом высокого разрешения (640x480, 16 цветов, 1 страница).

Не так давно появились так называемые супер-*VGA* адаптеры (*SVGA*) с разрешением 800x600 и более, использующие 256 и более цветовых оттенков. В настоящее время эти адаптеры получили повсеместное распространение, однако в библиотеке *Graph* для них нет драйверов. Поскольку *SVGA* совместимы с *VGA*, для управления современными графическими адаптерами приходится использовать драйвер *EGAVGA.BGI* и довольствоваться его относительно скромными возможностями.

Несколько особняком стоят достаточно популярные адаптеры фирмы *Hercules*. Адаптер *HGC* имеет разрешение 720x348, его пикселы могут светиться одним цветом (обычно светло-коричневым) или не светиться вовсе, т.е. это монохромный адаптер. Адаптер *HGC+* отличается несущественными усовершенствованиями, а адаптер *HICC* (*Hercules In Color Card*) представляет собой 16-цветный вариант *HGC+*.

14.1.2. Процедуры и функции

Процедура `InitGraph`. Иницирует графический режим работы адаптера. Заголовок процедуры:

Procedure `InitGraph`(**var** `Driver`, `Mode`: `Integer`; `Path`: `String`);

Здесь *Driver* — переменная типа *Integer*, определяет тип графического драйвера; *Mode* — переменная того же типа, задающая режим работы графического адаптера; *Path* — выражение типа *String*, содержащее имя файла драйвера и, возможно, маршрут его поиска.

К моменту вызова процедуры на одном из дисковых носителей информации должен находиться файл, содержащий нужный графический драйвер. Процедура загружает этот драйвер в оперативную память и переводит адаптер в графический режим работы. Тип драйвера должен соответствовать типу графического адаптера. Для указания типа драйвера в модуле предопределены следующие константы:

const

```
Detect      = 0; {Режим автоопределения типа}
CGA          = 1;
MCGA        = 2;
EGA          = 3;
EGA64       = 4;
EGAMono     = 5;
IBM8514     = 6;
HercMono    = 7;
ATT400      = 8;
VGA         = 9;
PC3270      = 10;
```

Большинство адаптеров могут работать в различных режимах. Для того, чтобы указать адаптеру требуемый режим работы, используется переменная *Mode*, значением которой в момент обращения к процедуре могут быть такие константы:

const

{Адаптер CGA:}

```
CGAC0       = 0; {Низкое разрешение, палитра 0}
CGAC1       = 1; {Низкое разрешение, палитра 1}
CGAC2       = 2; {Низкое разрешение, палитра 2}
CGAC3       = 3; {Низкое разрешение, палитра 3}
CGANi       = 4; {Высокое разрешение}
```

{Адаптер MCGA:}

```
MCGAC0      = 0; {Эмуляция CGAC0}
MCGAC1      = 1; {Эмуляция CGAC1}
MCGAC2      = 2; {Эмуляция CGAC2}
MCGAC3      = 3; {Эмуляция CGAC3}
MCGAMed     = 4; {Эмуляция CGANi}
MCGANi      = 5; {640x480}
```

```

{Адаптер EGA:}
EGALo      = 0; {640x200, 16 цветов}
EGANi      = 1; {640x350, 16 цветов}
EGAMonoHi  = 3; {640x350, 2 цвета}
{Адаптеры HGC и HGC+:}
HercMonoHi = 0; {720x348}
{Адаптер ATT400:}
ATT400C0   = 0; {Аналог режима CGAC0}
ATT400C1   = 1; {Аналог режима CGAC1}
ATT400C2   = 2; {Аналог режима CGAC2}
ATT400C3   = 3; {Аналог режима CGAC3}
ATT400Med  = 4; {Аналог режима CGANi}
ATT400Hi   = 5; {640x400, 2 цвета}
{Адаптер VGA:}
VGA Lo     = 0; {640x200}
VGAMed     = 1; {640x350}
VGANi      = 2; {640x480}
PC3270Hi   = 0; {Аналог HercMonoHi}
{Адаптер IBM8514}
IBM8514Lo  = 0; {640x480, 256 цветов}
IBM8514Hi  = 1; {1024x768, 256 цветов}

```

Пусть, например, драйвер *CGA.BGI* находится в каталоге *TP\BGI* на диске *C* и устанавливается режим работы 320x200 с палитрой 2. Тогда обращение к процедуре будет таким:

```

Uses Graph;
var
  Driver, Mode : Integer;
begin
  Driver := CGA;           {Драйвер}
  Mode   := CGAC2;        {Режим работы}
  InitGraph(Driver, Mode, 'C:\TP\BGI');
  .....

```

Если тип адаптера ПК неизвестен или если программа рассчитана на работу с любым адаптером, используется обращение к процедуре с требованием автоматического определения типа драйвера:

```

Driver := Detect;
InitGraph(Driver, Mode, 'C:\TP\BGI');

```

После такого обращения устанавливается графический режим работы экрана, а при выходе из процедуры переменные *Driver* и *Mode* содержат целочисленные значения, определяющие тип драйвера и режим его работы. При этом для адаптеров, способных работать в нескольких режимах, выбирается старший режим, т.е. тот, что закодирован максимальной цифрой. Так, при работе с *CGA*-адаптером обращение к процедуре со значением *Driver = Detect* вернет в переменной *Driver* значение 1 (*CGA*) и

в *Mode* — значение 4 (CGAHi), а такое же обращение к адаптеру VGA вернет *Driver* = 9 (VGA) и *Mode* = 2 (VGAHi).

Функция *GraphResult*. Возвращает значение типа *Integer*, в котором закодирован результат последнего обращения к графическим процедурам. Если ошибка не обнаружена, значением функции будет ноль, в противном случае — отрицательное число, имеющее следующий смысл:

const

```
grOk           = 0;    {Нет ошибок}
grInitGraph    = -1;   {Не инициирован графический режим}
grNotDetected  = -2;   {Не определен тип драйвера}
grFileNotFound = -3;   {Не найден графический драйвер}
grInvalidDriver = -4;  {Неправильный тип драйвера}
grNoLoadMem    = -5;   {Нет памяти для размещения драйвера}
grNoScanMem    = -6;   {Нет памяти для просмотра областей}
grNoFloodMem   = -7;   {Нет памяти для закрашки областей}
grFontNotFound = -8;   {Не найден файл со шрифтом}
grNoFontMem    = -9;   {Нет памяти для размещения шрифта}
grInvalidMode  = -10;  {Неправильный графический режим}
grError        = -11;  {Общая ошибка}
grIOError      = -12;  {Ошибка ввода-вывода}
grInvalidFont  = -13;  {Неправильный формат шрифта}
grInvalidFontNum = -14; {Неправильный номер шрифта}
```

После обращения к функции *GraphResult* признак ошибки сбрасывается, поэтому повторное обращение к ней вернет ноль.

Функция *GraphErrorMsg*. Возвращает значение типа *String*, в котором по указанному коду ошибки дается соответствующее текстовое сообщение. Заголовок функции:

Function *GraphErrorMsg*(Code: Integer): String;

Здесь *Code* — код ошибки, возвращаемый функцией *GraphResult*.

Например, типичная последовательность операторов для инициации графического режима с автоматическим определением типа драйвера и установкой максимального разрешения имеет следующий вид:

var

Driver, *Mode*, *Error* : Integer;

begin

```
Driver := Detect;           {Автоопределение драйвера}
InitGraph(Driver, Mode, ''); {Инициуем графику}
Error := GraphResult;      {Получаем результат}
if Error <> grOk then       {Проверяем ошибку}
  begin                   {Ошибка в процедуре инициации}
    WriteLn(GraphErrorMsg(Error)); {Выводим сообщение}
    .....
  end
else                      {Нет ошибки}
```

.....

Чаще всего причиной возникновения ошибки при обращении к процедуре *InitGraph* является неправильное указание местоположения файла с драйвером графического адаптера (например, файла *CGA.BGI* для адаптера *CGA*). Настройка на местоположение драйвера осуществляется заданием маршрута поиска нужного файла в имени драйвера при вызове процедуры *InitGraph*. Если, например, драйвер зарегистрирован в подкаталоге *DRIVERS* каталога *PASCAL* на диске *D*, то нужно использовать вызов:

```
InitGraph(Driver, Mode, 'd:\Pascal\Drivers');
```

Замечание. Во всех следующих примерах процедура *InitGraph* вызывается с параметром *Driver* в виде пустой строки. Такая форма обращения будет корректна только в том случае, когда нужный файл графического драйвера находится в текущем каталоге. Для упрощения повторения примеров скопируйте файл, соответствующий адаптеру Вашего ПК, в текущий каталог.

Процедура *CloseGraph*. Завершает работу адаптера в графическом режиме и восстанавливает текстовый режим работы экрана. Заголовок:

```
Procedure CloseGraph;
```

Процедура *RestoreCRTMode*. Служит для кратковременного возврата в текстовый режим. В отличие от процедуры *CloseGraph* не сбрасываются установленные параметры графического режима и не освобождается память, выделенная для размещения графического драйвера. Заголовок:

```
Procedure RestoreCRTMode;
```

Функция *GetGraphMode*. Возвращает значение типа *Integer*, в котором содержится код установленного режима работы графического адаптера. Заголовок:

```
Function GetGraphMode: Integer;
```

Процедура *SetGraphMode*. Устанавливает новый графический режим работы адаптера. Заголовок:

```
Procedure SetGraphMode(Mode: Integer);
```

Здесь *Mode* — код устанавливаемого режима.

Следующая программа иллюстрирует переход из графического режима в текстовый и обратно:

```
uses Graph;
var
  Driver, Mode, Error : Integer;
begin
  {Инициализируем графический режим}
  Driver := Detect;
```

```

InitGraph(Driver, Mode, '');
Error := GraphResult;           {Запоминаем результат}
if Error <> grOk then            {Проверяем ошибку}
  WriteLn(GraphErrorMsg(Error)) {Есть ошибка}
else
  begin                          {Нет ошибки}
    WriteLn ('Это графический режим');
    WriteLn ('Нажмите "Enter"...':20);
    ReadLn;
    {Переходим в текстовый режим}
    RestoreCRTMode;
    WriteLn (' А это текстовый...');
    ReadLn;
    {Возвращаемся в графический режим}
    SetGraphMode (GetGraphMode);
    WriteLn ('Опять графический режим...');
    ReadLn;
    CloseGraph
  end
end.

```

В этом примере для вывода сообщений как в графическом, так и в текстовом режиме используется стандартная процедура *WriteLn*. Если Ваш ПК оснащен нерусифицированным адаптером CGA, вывод кириллицы в графическом режиме таким способом невозможен, в этом случае замените соответствующие сообщения так, чтобы использовать только латинские буквы.

Процедура DetectGraph. Возвращает тип драйвера и режим его работы. Заголовок:

```
Procedure DetectGraph(var Driver, Mode: Integer);
```

Здесь *Driver* — тип драйвера; *Mode* — режим работы.

В отличие от функции *GetGraphMode* описываемая процедура возвращает в переменную *Mode* максимально возможный для данного адаптера номер графического режима.

Функция GetDriverName. Возвращает значение типа *String*, содержащее имя загруженного графического драйвера. Заголовок:

```
Function GetDriverName: String;
```

Функция GetMaxMode. Возвращает значение типа *Integer*, содержащее количество возможных режимов работы адаптера. Заголовок:

```
Function GetMaxMode: Integer;
```

Функция GetModeName. Возвращает значение типа *String*, содержащее разрешение экрана и имя режима работы адаптера по его номеру. Заголовок:

```
Function GetModName (ModNumber: Integer): String;
```

Здесь *ModNumber* — номер режима.

Следующая программа после инициации графического режима выводит на экран строку, содержащую имя загруженного драйвера, а также все возможные режимы его работы.

```
Uses Graph;
var
  a,b: Integer;
begin
  a := Detect;
  InitGraph(a, b, '');
  WriteLn(GetDriverName);
  for a := 0 to GetMaxMode do
    WriteLn(GetModeName(a):10);
  ReadLn;
  CloseGraph
end.
```

Процедура *GetModeRange*. Возвращает диапазон возможных режимов работы заданного графического адаптера. Заголовок:

```
Procedure GetModeRange (Drv: Integer; var Min, Max: Integer);
```

Здесь *Drv* — тип адаптера; *Min* — переменная типа *Integer*, в которой возвращается нижнее возможное значение номера режима; *Max* — переменная того же типа, верхнее значение номера.

Если задано неправильное значение параметра *Drv*, процедура вернет в обеих переменных значение -1 . Перед обращением к процедуре можно не устанавливать графический режим работы экрана. Следующая программа выводит на экран названия всех адаптеров и диапазоны возможных номеров режимов их работы.

```
Uses Graph;
var
  D,L,H: Integer;
const
  N: array [1..11] of String [8] =
    ('CGA      ', 'MCGA     ', 'EGA       ',
     'EGA64     ', 'EGAMono   ', 'IBM8514   ',
     'HercMono', 'ATT400    ', 'VGA       ',
     'PC3270    ', 'Ошибка   ');
begin
  WriteLn('Адаптер      Мин.      Макс. ');
  for D := 1 to 11 do
    begin
      GetModeRange(D, L, H);
      WriteLn(N[D], L:7, H:10)
    end
```

end.

14.2. КООРДИНАТЫ, ОКНА, СТРАНИЦЫ

Многие графические процедуры и функции используют указатель текущей позиции на экране, который в отличие от текстового курсора невидим. Положение этого указателя, как и вообще любая координата на графическом экране, задается относительно левого верхнего угла, который, в свою очередь, имеет координаты 0,0. Таким образом, горизонтальная координата экрана увеличивается слева направо, а вертикальная — сверху вниз.

Функции *GetMaxX* и *GetMaxY*. Возвращают значения типа *Word*, со — держащие максимальные координаты экрана в текущем режиме работы соответственно по горизонтали и вертикали. Например:

```
Uses Graph;  
var  
  a,b: Integer;  
begin  
  a := Detect;  
  InitGraph(a, b, '');  
  WriteLn(GetMaxX, GetMaxY:5);  
  ReadLn;  
  CloseGraph  
end.
```

Функции *GetX* и *GetY*. Возвращают значения типа *Integer*, содержащие текущие координаты указателя соответственно по горизонтали и вертика — ли. Координаты определяются относительно левого верхнего угла окна или, если окно не установлено, экрана.

Процедура *SetViewPort*. Устанавливает прямоугольное окно на гра — фическом экране. Заголовок:

```
Procedure SetViewPort (X1,Y1,X2,Y2 Integer; ClipOn: Boolean);
```

Здесь *X1...Y2* — координаты левого верхнего (*X1,Y1*) и правого нижнего (*X2,Y2*) углов окна; *ClipOn* — выражение типа *Boolean*, определяющее «отсечку» не умещающихся в окне элементов изображения.

Координаты окна всегда задаются относительно левого верхнего угла экрана. Если параметр *ClipOn* имеет значение *True*, элементы изображе — ния, не умещающиеся в пределах окна, отсекаются, в противном случае границы окна игнорируются. Для управления этим параметром можно использовать такие определенные в модуле константы:

```
const  
  ClipOn = True;           {Включить отсечку}  
  ClipOff = False;        {Не включать отсечку}
```


Следующий пример иллюстрирует действие параметра *ClipOn*. Программа строит два прямоугольных окна с разными значениями параметра и выводит в них несколько окружностей. Для большей наглядности окна обводятся рамками (см. рис.14.1).

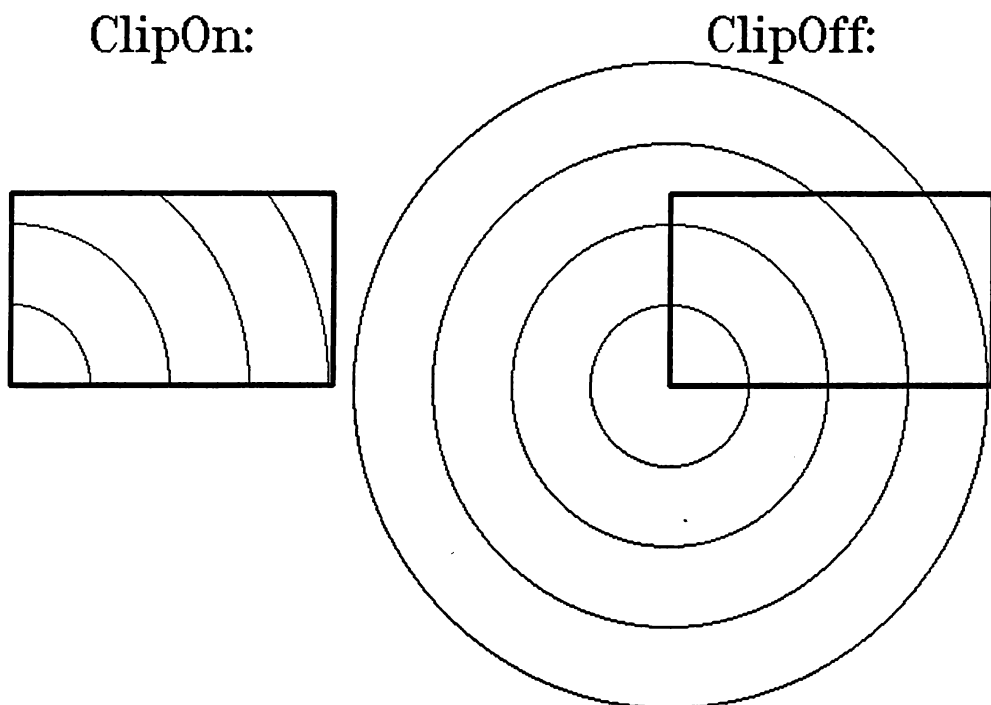


Рис.14.1. Отсечка изображения в окне

```

Uses Graph, CRT;
var
  x, y, e: Integer;
  x11, y11, x12, y12,           {Координаты 1-го окна}
  x21, x22,                     {Левый верхний угол 2-го}
  R,                             {Начальный радиус}
  k: Integer;
begin
  DirectVideo := False;          {Блокируем прямой доступ
                                  к видеопамати в модуле CRT}
  {Иницилируем графический режим}
  x := Detect;
  InitGraph(x, y, '');
  {Проверяем результат}
  e := GraphResult;
  if e <> grOk then
    WriteLn(GraphErrorMsg(e))    {Ошибка}
  else

```

```

begin                                     {Нет ошибки}
  {Вычисляем координаты с учетом разрешения экрана}
  x11 := GetMaxX div 60;
  x12 := GetMaxX div 3;
  y11 := GetMaxY div 4;
  y12 := 2*y11;
  R   := (x12-x11) div 4;
  x21 := x12*2;
  x22 := x21+x12-x11;
  {Рисуем окна}
  WriteLn('ClipOn:':10,'ClipOff:':40);
  Rectangle(x11, y11, x12, y12);
  Rectangle(x21, y11, x22, y12);
  {Назначаем 1-е окно и рисуем четыре окружности}
  SetViewport(x11, y11, x12, y12, ClipOn);
  for k := 1 to 4 do
    Circle(0,y11,R*k);
  {Назначаем 2-е окно и рисуем окружности}
  SetViewport(x21, y11, x22, y12, ClipOff);
  for k := 1 to 4 do
    Circle(0,y11,R*k);
  {Ждем нажатия любой клавиши}
  if ReadKey=#0 then k := ord(ReadKey);
  CloseGraph
end
end.

```

Процедура GetViewSettings. Возвращает координаты и признак отсечки текущего графического окна. Заголовок:

Procedure GetViewSettings(**var** ViewInfo: ViewPortType);

Здесь *ViewInfo* — переменная типа *ViewPortType*. Этот тип в модуле *Graph* определен следующим образом:

```

type
  ViewPortType = record
    x1,y1,x2,y2: Integer; {Координаты окна}
    Clip       : Boolean  {Признак отсечки}
  end;

```

Процедура MoveTo. Устанавливает новое текущее положение указателя. Заголовок:

Procedure MoveTo(X,Y: Integer);

Здесь *X,Y* — новые координаты указателя соответственно по горизонтали и вертикали.

Координаты определяются относительно левого верхнего угла окна или, если окно не установлено, экрана.

Процедура MoveRel. Устанавливает новое положение указателя в от-носительных координатах.

Procedure MoveRel (DX,DY: Integer);

Здесь DX,DY — приращения новых координат указателя соответственно по горизонтали и вертикали.

Приращения задаются относительно того положения, которое занимал указатель к моменту обращения к процедуре.

Процедура ClearDevice. Очищает графический экран. После обра-щения к процедуре указатель устанавливается в левый верхний угол эк-рана, а сам экран заполняется цветом фона, заданным процедурой *SetBkColor*. Заголовок:

Procedure ClearDevice;

Процедура ClearViewPort. Очищает графическое окно, а если окно не определено к этому моменту — весь экран. При очистке окно заполняется цветом с номером 0 из текущей палитры. Указатель перемещается в левый верхний угол окна. Заголовок:

Procedure ClearViewPort;

В следующей программе на экране создается окно, которое затем за-полняется случайными окружностями (рис.14.2). После нажатия на любую клавишу окно очищается. Для выхода из программы нажмите *Enter*.

Uses CRT, Graph;

var

x1,y1,x2,y2,Err: Integer;

begin

{Иницилируем графический режим}

x1 := Detect;

InitGraph(x1,x2,'');

Err := GraphResult;

if Err<>grOk **then**

WriteLn (GraphErrorMsg (Err))

else

begin

{Определяем координаты окна

с учетом разрешения экрана}

x1 := GetMaxX **div** 4;

y1 := GetMaxY **div** 4;

x2 := 3*x1;

y2 := 3*y1;

{Создаем окно}

Rectangle(x1,y1,x2,y2);

SetViewPort(x1+1,y1+1,x2-1,y2-1,ClipOn);

{Заполняем окно случайными окружностями}

repeat

```

    Circle(Random(GetMaxX),Random(GetMaxX),
           Random(GetMaxX div 5))
until KeyPressed;

```

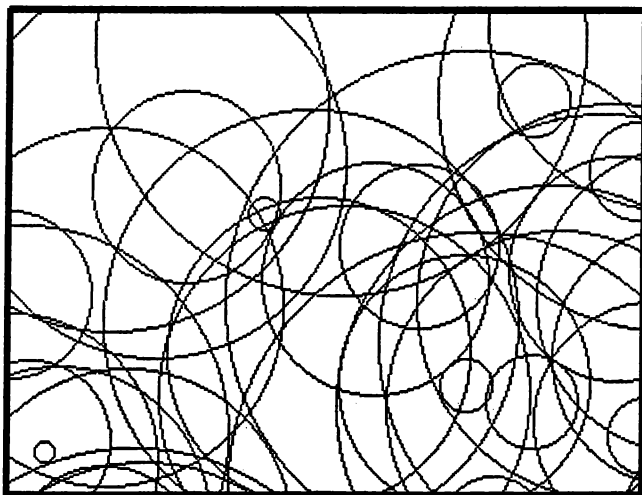


Рис.14.2. Окно со случайными окружностями

```

{Очищаем окно и ждем нажатия Enter}
  ClearViewPort;
  OutTextXY(0,0,'Press Enter...');
  ReadLn;
  CloseGraph
end
end.

```

Процедура GetAspectRatio. Возвращает два числа, позволяющие оценить соотношение сторон экрана. Заголовок:

Procedure GetAspectRatio(**var** X,Y: Word);

Здесь X, Y — переменные типа Word. Значения, возвращаемые в этих переменных, позволяют вычислить отношение сторон графического экрана в пикселах. Найденный с их помощью коэффициент может использоваться при построении правильных геометрических фигур, таких как окружности, квадраты и т.п. Например, если Вы хотите построить квадрат со стороной 1 пиксел по вертикали, Вы должны использовать операторы

```

GetAspectRatio (Xasp, Yasp);
Rectangle(x1, y1, x1+L*round (Yasp/Xasp), y1+L);

```

а если L определяет длину квадрата по горизонтали, то используется оператор

```

Rectangle(x1,y1,x1+L,y1+L*round(Xasp/Yasp));

```

Процедура *SetAspectRatio*. Устанавливает масштабный коэффициент отношения сторон графического экрана. Заголовок:

Procedure *SetAspectRatio*(X,Y: Word);

Здесь X, Y — устанавливаемые соотношения сторон.

Следующая программа строит 20 окружностей с разными соотношениями сторон экрана (рис.14.3).

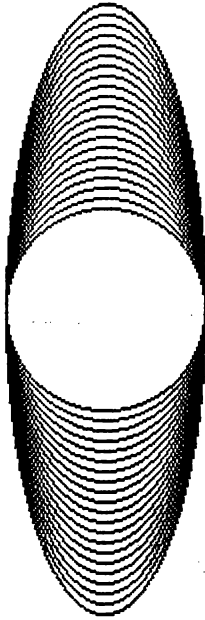


Рис.14.3. Окружности при разных отношениях сторон экрана

```
Uses Graph,CRT;
const
  R = 50;
  dx = 1000;
var
  d,m,e,k : Integer;
  Xasp,Yasp: Word;
begin
  d := Detect;
  InitGraph(d, m, '');
  e := GraphResult;
  if e <> grOk then
    WriteLn(GraphErrorMsg(e))
  else
    begin
      GetAspectRatio(Xasp, Yasp);
      for k := 0 to 20 do
```

```

begin
  SetAspectRatio (Xasp+k*dx,Yasp);
  Circle (GetMaxX div 2,GetMaxY div 2,R)
end;
if ReadKey=#0 then k := ord(ReadKey);
CloseGraph
end
end.

```

Процедура SetActivePage. Делает активной указанную страницу видеопамати. Заголовок:

```
Procedure SetActivePage (PageNum: Word);
```

Здесь *PageNum* — номер страницы.

Процедура может использоваться только с адаптерами, поддерживающими многостраничную работу (EGA, VGA и т.п.). Фактически процедура просто переадресует графический вывод в другую область видеопамати, однако вывод текстов с помощью *Write/WriteLn* всегда осуществляется только на страницу, которая является видимой в данный момент (активная страница может быть невидимой). Нумерация страниц начинается с нуля.

Процедура SetVisualPage. Делает видимой страницу с указанным номером. Обращение:

```
Procedure SetVisualPage (PageNum: Word);
```

Здесь *PageNum* — номер страницы.

Процедура может использоваться только с адаптерами, поддерживающими многостраничную работу (EGA, VGA и т.п.). Нумерация страниц начинается с нуля.

Следующая программа сначала рисует квадрат в видимой странице и окружность — в невидимой. После нажатия на *Enter* происходит смена видимых страниц.

```
Uses Graph;
```

```
var
```

```
  d,m,e: Integer;
```

```
  s      : String;
```

```
begin
```

```
  d := Detect;
```

```
  InitGraph(d, m, '');;
```

```
  e := GraphResult;
```

```
  if e <> grOk then
```

```
    WriteLn (GraphErrorMsg(e))
```

```
  else {Нет ошибки. Проверяем, поддерживает ли драйвер
        многостраничную работу с видеопаматью:}
```

```
  if d in [HercMono,EGA,EGA64,MCGA,VGA] then
```

```
    begin {Используем многостраничный режим}
```

```

if d<>HercMono then
    SetGraphMode(m-1);
    {Заполняем видимую страницу}
    Rectangle(10,10,GetMaxX div 2,GetMaxY div 2);
    OutTextXY(0,0,'Page 0. Press Enter...');
    {Заполняем невидимую}
    SetActivePage(1);
    Circle(GetMaxX div 2, GetMaxY div 2, 100);
    OutTextXY(0,GetMaxY-10,'Page 1. Press Enter...');
    {Демонстрируем страницы}
    ReadLn;
    SetVisualPage(1);
    ReadLn;
    SetVisualPage(0);
    ReadLn;
    CloseGraph
end
else
    begin      {Драйвер не поддерживает многостраничный режим}
        s := GetDriverName;
        CloseGraph;
        WriteLn('Адаптер ',s,' использует только 1 страницу')
    end
end.

```

Обратите внимание на оператор

```

if d<>HercMono then
    SetGraphMode(m-1);

```

С его помощью гарантированно устанавливается многостраничный режим работы на адаптерах EGA, MCGA, VGA. Как уже говорилось, после инициализации графики с *Driver=Detect* устанавливается режим работы с максимально возможным номером; перечисленные адаптеры в этом режиме могут работать только с одной графической страницей, чтобы обеспечить работу с двумя страницами, следует уменьшить номер режима.

13.3. ЛИНИИ И ТОЧКИ

Процедура PutPixel. Выводит заданным цветом точку по указанным координатам. Заголовок:

```
Procedure PutPixel(X,Y: Integer; Color: Word);
```

Здесь X,Y — координаты точки; Color — цвет точки.

Координаты задаются относительно левого верхнего угла окна или, если окно не установлено, относительно левого верхнего угла экрана.

Следующая программа периодически выводит на экран «звездное небо» и затем гасит его. Для выхода из программы нажмите любую клавишу.

```

Uses CRT, Graph;
type
  PixelType = record
    x, y : Integer;
  end;
const
  N = 5000;                                {Количество "звезд"}
var
  d,r,e,k: Integer;
  x1,y1,x2,y2: Integer;
  a: array [1..N] of PixelType;          {Координаты}
begin
  {Иницилируем графику}
  d := Detect;
  InitGraph(d, r, '');
  e := GraphResult;
  if e<>grOk then
    WriteLn(GraphErrorMsg(e))
  else
    begin
      {Создаем окно в центре экрана}
      x1 := GetMaxX div 4;
      y1 := GetMaxY div 4;
      x2 := 3*x1;
      y2 := 3*y1;
      Rectangle(x1,y1,x2,y2);
      SetViewport(x1+1,y1+1,x2-1,y2-1,ClipOn);
      {Создаем и запоминаем координаты всех "звезд"}
      for k := 1 to N do
        with a[k] do
          begin
            x := Random(x2-x1);
            y := Random(y2-y1)
          end;
        {Цикл вывода}
      repeat
        for k := 1 to N do
          with a[k] do                                {Зажигаем "звезду"}
            PutPixel(x,y,white);
          if not KeyPressed then
            for k := N downto 1 do
              with a[k] do                                {Гасим "звезду"}
                PutPixel(x,y,black)
            until KeyPressed;
          while KeyPressed do k := ord(ReadKey);
        CloseGraph
      end;
    end;
  end;

```


end.

Функция *GetPixel*. Возвращает значение типа *Word*, содержащее цвет пиксела с указанными координатами. Заголовок:

Function *GetPixel* (X,Y: Integer): Word;

Здесь X, Y — координаты пиксела.

Процедура *Line*. Вычерчивает линию с указанными координатами начала и конца. Заголовок:

Procedure *Line* (X1,Y1,X2,Y2: Integer);

Здесь X1...Y1 — координаты начала (X1, Y1) и конца (X2, Y2) линии.

Линия вычерчивается текущим стилем и текущим цветом. В следующей программе в центре экрана создается окно, которое затем расчерчивается случайными линиями. Для выхода из программы нажмите любую клавишу.

```
Uses CRT, Graph;
var
  d,r,e : Integer;
  x1,y1,x2,y2: Integer;
begin
  {Иницилируем графику}
  d := Detect;
  InitGraph(d, r, '');
  e := GraphResult;
  if e <> grOk then
    WriteLn(GraphErrorMsg(e))
  else
    begin
      {Создаем окно в центре экрана}
      x1 := GetMaxX div 4;
      y1 := GetMaxY div 4;
      x2 := 3*x1;
      y2 := 3*y1;
      Rectangle(x1,y1,x2,y2);
      SetViewport(x1+1,y1+1,x2-1,y2-1,ClipOn);
      {Цикл вывода случайных линий}
      repeat
        SetColor(succ(Random(16))); {Случайный цвет}
        Line(Random(x2-x1), Random(y2-y1),
              Random(x2-x1), Random(y2-y1))
      until KeyPressed;
      if ReadKey=#0 then d := ord(ReadKey);
      CloseGraph
    end
  end.
end.
```

Процедура LineTo. Вычерчивает линию от текущего положения указателя до положения, заданного его новыми координатами. Заголовок:

Procedure LineTo(*X*,*Y*: Integer);

Здесь *X*, *Y* — координаты нового положения указателя, они же — координаты второго конца линии.

Процедура LineRel. Вычерчивает линию от текущего положения указателя до положения, заданного приращениями его координат. Заголовок:

Procedure LineRel (*DX*, *DY*: Integer);

Здесь *DX*, *DY* — приращения координат нового положения указателя.

В процедурах *LineTo* и *LineRel* линия вычерчивается текущим стилем и текущим цветом.

Процедура SetLineStyle. Устанавливает новый стиль вычерчиваемых линий. Заголовок:

Procedure SetLineStyle(**Type**, *Pattern*, *Thick*: Word)

Здесь *Type*, *Pattern*, *Thick* — соответственно тип, образец и толщина линии.

Тип линии может быть задан с помощью одной из следующих констант:

```
const
    SolidLn      = 0; {Сплошная линия}
    DottedLn     = 1; {Точечная линия}
    CenterLn     = 2; {Штрих-пунктирная линия}
    DashedLn     = 3; {Пунктирная линия}
    UserBitLn    = 4; {Узор линии определяет пользователь}
```

Параметр *Pattern* учитывается только для линий, вид которых определяется пользователем (т.е. в случае, когда *Type* = *UserBitLn*). При этом два байта параметра *Pattern* определяют образец линии: каждый установленный в единицу бит этого слова соответствует светящемуся пикселу в линии, нулевой бит — несветящемуся пикселу. Таким образом, параметр *Pattern* задает отрезок линии длиной в 16 пиксел. Этот образец периодически повторяется по всей длине линии.

Параметр *Thick* может принимать одно из двух значений:

```
const
    NormWidth    = 1; {Толщина в один пиксел}
    ThickWidth   = 3; {Толщина в три пиксела}
```

Отметим, что установленный процедурой стиль линий (текущий стиль) используется при построении прямоугольников, многоугольников и других фигур.

В следующем примере демонстрируются линии всех стандартных стилей, затем вводятся слово-образец и линия с этим образцом заполнения (рис.14.4). Для выхода из программы введите ноль.

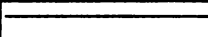
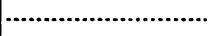
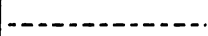
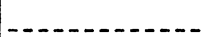
	SolidLn	Pattern: 1	
	DottedLn	Pattern: 7	
	CenterLn	Pattern: 127	
	DashedLn	Pattern: 65520	
	SolidLn	Pattern:	
	DottedLn		
	CenterLn		
	DashedLn		

Рис.14.4. Образцы линий

```

Uses CRT, Graph;
const
  style: array [0..4] of String [9] = (
    'SolidLn ', 'DottedLn ', 'CenterLn',
    'DashedLn', 'UserBitLn');
var
  d,r,e,i,j,dx,dy: Integer;
  p: Word;
begin
  {Иницилируем графику}
  d := Detect;
  InitGraph(d, r, '');
  e := GraphResult;
  if e <> grOk then
    WriteLn (GraphErrorMsg(e))
  else
    begin
      {Вычисляем смещение линий}
      dx := GetMaxX div 6;
      dy := GetMaxY div 10;
      {Выводим стандартные линии}
      for j := 0 to 1 do                                {Для двух толщин}
        begin
          for i := 0 to 3 do                                {Четыре типа линий}
            begin
              SetLineStyle(i, 0, j*2+1);
              Line(0, (i+j*4+1)*dy,dx, (i+j*4+1)*dy);
              OutTextXY(dx+10, (i+j*4+1)*dy, style[i])
            end
          end;
        {Вводим образец и чертим линию.}
        j := 0;
        dy := (GetMaxY+1) div 25;

```

```

repeat
  OutTextXY(320,j*dy,'Pattern: ');
  GotoXY(50,j+1);
  ReadLn(p);
  if p <> 0 then
    begin
      SetLineStyle(UserBitLn,p,NormWidth);
      Line(440,j*dy+4, 600, j*dy+4);
      inc(j)
    end
  until p = 0;
  CloseGraph
end
end.

```

Процедура GetLineSettings. Возвращает текущий стиль линий. Заголовок:

```
Procedure GetLineSettings(var StyleInfo: LineSettingsType)
```

Здесь *StyleInfo* — переменная типа *LineSettingsType*, в которой возвращается текущий стиль линий.

Тип *LineSettingsType* определен в модуле *Graph* следующим образом:

```

type
LineSettingsType = record
  LineStyle: Word; {Тип линии}
  Pattern : Word; {Образец}
  Thickness: Word {Толщина}
end;

```

Процедура SetWriteMode. Устанавливает способ взаимодействия вновь выводимых линий с уже существующим на экране изображением. Заголовок:

```
Procedure SetWriteMode (Mode);
```

Здесь *Mode* — выражение типа *Integer*, задающее способ взаимодействия выводимых линий с изображением.

Если параметр *Mode* имеет значение 0, выводимые линии накладываются на существующее изображение обычным образом (инструкцией *MOV* центрального процессора). Если значение 1, то это наложение осуществляется с применением логической операции **XOR** (исключительное ИЛИ): в точках пересечения выводимой линии с имеющимся на экране изображением светимость пиксел инвертируется на обратную, так что два следующих друг за другом вывода одной и той же линии на экран не изменят его вид.

Режим, установленный процедурой *SetWriteMode*, распространяется на процедуры *Drawpoly*, *Line*, *LineRel*, *LineTo* и *Rectangle*. Для задания пара-

метра *Mode* можно использовать следующие определенные в модуле константы:

const

```
CopyPut = 0; {Наложение операцией MOV}
XORPut  = 1; {Наложение операцией XOR}
```

В следующем примере на экране имитируется вид часового циферблата (рис.14.5). Для наглядной демонстрации темп хода «часов» ускорен в 600 раз (см. оператор *Delay(100)*). При желании Вы сможете легко усложнить программу, связав ее показания с системными часами и добавив секундную стрелку. Для выхода из программы нажмите на любую клавишу.

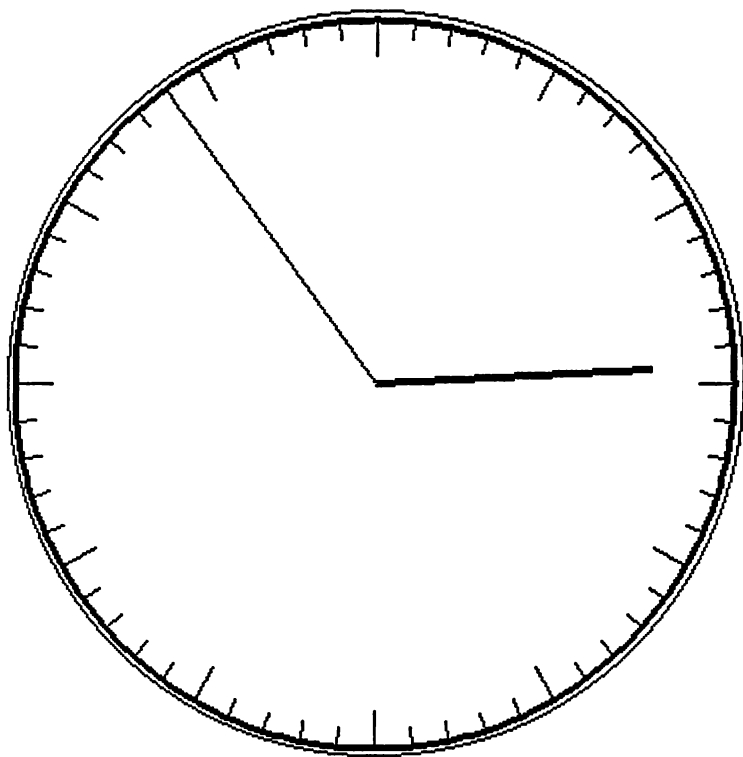


Рис.14.5. Часовой циферблат

```
Uses Graph, CRT;
var
  d,r,r1,r2,rr,k,
  x1,y1,x2,y2,x01,y01: Integer;
  Xasp,Yasp : Word;
begin
  {Инициализируем графику}
  d := detect;
```

```

InitGraph(d, r, '');
k := GraphResult;
if k <> grOK then
  WriteLn(GraphErrorMsg(k))
else
  begin
    {Определяем отношение сторон и размеры экрана}
    x1 := GetMaxX div 2;
    y1 := GetMaxY div 2;
    GetAspectRatio(Xasp, Yasp);
    {Вычисляем радиусы:}
    r := round(3*GetMaxY*Yasp/8/Xasp);
    r1 := round(0.9*r);      {Часовые деления}
    r2 := round(0.95*r);    {Минутные деления}
    {Изображаем циферблат}
    Circle(x1,y1,r);        {Первая внешняя окружность}
    Circle(x1,y1,round(1.02*r)); {Вторая окружность}
    for k := 0 to 59 do      {Деления циферблата}
      begin
        if k mod 5=0 then
          rr := r1           {Часовые деления}
        else
          rr := r2;          {Минутные деления}
        {Определяем координаты концов делений}
        x01 := x1+Round(rr*sin(2*pi*k/60));
        y01 := y1-Round(rr*Xasp*cos(2*pi*k/60)/Yasp);
        x2 := x1+Round(r*sin(2*pi*k/60));
        y2 := y1-Round(r*Xasp*cos(2*pi*k/60)/Yasp);
        Line(x01,y01,x2,y2)    {Выводим деление}
      end;
      {ГОТОВИМ ВЫВОД СТРЕЛОК}
      SetWriteMode(XORPut);
      SetLineStyle(SolidLn,0,ThickWidth);
      r := 0;                  {Счетчик минут в одном часе}
      {Цикл вывода стрелок}
      repeat
        for k := 0 to 59 do    {k = минуты}
          if not KeyPressed then
            begin
              {Координаты часовой стрелки}
              x2 := x1+Round(0.85*r1*sin(2*pi*r/60/12));
              y2 := y1-Round(0.85*r1*Xasp*cos(2*pi*r/60/12)/Yasp);
              {Координаты минутной стрелки}
              x01 := x1+Round(r2*sin(2*pi*k/60));
              y01 := y1-Round(r2*Xasp*cos(2*pi*k/60)/Yasp);
              {Изображаем стрелки}
              Line(x1,y1,x2,y2);
              Line(x1,y1,x01,y01);
            end;
          end;
        end;
      end;
    end;
  end;

```

```

Delay(100);      {Для имитации реального темпа
                  нужно установить задержку 60000}
{Для удаления стрелок выводим их еще раз!}
Line(x1,y1,x01,y01);
Line(x1,y1,x2,y2);
{Наращиваем и корректируем счетчик минут в часе}
inc(r);
if r=12*60 then
  r := 0
end
until KeyPressed;
if ReadKey=#0 then k := ord(ReadKey);
CloseGraph
end
end.

```

14.4. МНОГОУГОЛЬНИКИ

Процедура *Rectangle*. Вычерчивает прямоугольник с указанными координатами углов. Заголовок:

Procedure *Rectangle* (X1,Y1,X2,Y2: Integer);

Здесь X1...Y2 — координаты левого верхнего (X1, Y1) и правого нижнего (X2, Y2) углов прямоугольника. Прямоугольник вычерчивается с использованием текущего цвета и текущего стиля линий.

В следующем примере на экране вычерчиваются 10 вложенных друг в друга прямоугольников.

Uses Graph, CRT;

```

var
  d,r,e,x1,y1,
  x2,y2,dx,dy: Integer;
begin
  {Иницилируем графику}
  d := Detect;
  InitGraph(d, r, '');
  e := GraphResult;
  if e <> grOK then
    WriteLn(GraphErrorMsg(e))
  else
    begin
      {Определяем приращения сторон}
      dx := GetMaxX div 20;
      dy := GetMaxY div 20;
      {Чертим вложенные прямоугольники}
      for d := 0 to 9 do
        Rectangle(d*dx,d*dy,GetMaxX-d*dx,GetMaxY-d*dy);
        if ReadKey=#0 then d := ord(ReadKey);
      end
    end
  end
end.

```

```

    CloseGraph
  end
end.

```

Процедура *DrawPoly*. Вычерчивает произвольную ломаную линию, заданную координатами точек излома.

```

Procedure DrawPoly(N: Word; var Points)

```

Здесь *N* — количество точек излома, включая обе крайние точки; *Points* — переменная типа *PointType*, содержащая координаты точек излома.

Координаты точек излома задаются парой значений типа *Word*: первое определяет горизонтальную, второе — вертикальную координаты. Для них можно использовать следующий определенный в модуле тип:

```

type
  PointType = record
    x, y : Word
  end;

```

При вычерчивании используется текущий цвет и текущий стиль линий. Вот как, например, можно с помощью этой процедуры вывести на экран график синуса:

```

Uses Graph;
const
  N = 100; {Количество точек графика}
var
  d, r, e: Integer;
  m      : array [0..N+1] of PointType;
  k      : Word;
begin
  {Иницилируем графику}
  d := Detect;
  InitGraph(d, r, '');
  e := GraphResult;
  if e <> grOk then
    WriteLn(GraphErrorMsg(e))
  else
    begin
      {Вычисляем координаты графика}
      for k := 0 to N do with m[k] do
        begin
          x := trunc(k*GetMaxX/N);
          y := trunc(GetMaxY*(-sin(2*Pi*k/N)+1)/2)
        end;
      {Замыкаем график прямой линией}
      m[succ(N)].x := m[0].x;
      m[succ(n)].y := m[0].y;
      DrawPoly(N + 2, m);
    end;
  end;

```



```
    ReadLn;  
    CloseGraph  
end  
end.
```

В этом примере для проведения горизонтальной прямой используется «замыкание» ломаной — первая и последняя координаты ее точек излома совпадают.

Замечу, что хотя количество точек излома N — выражение типа *Word*, на самом деле внутри процедуры на этот параметр накладываются ограничения, связанные с конечным размером используемой буферной памяти. Вы можете убедиться в этом с помощью, например, изменения N в предыдущем примере: при $N=678$ график перестанет выводиться на экран, а функция *GraphResult* будет возвращать значение -6 (не хватает памяти для просмотра областей). Таким образом, для этой программы пороговое значение количества точек излома составляет 679. В то же время для программы

```
Uses Graph;  
const  
    N=510; {Предельное значение, при котором  
           на экране еще видна диагональная линия}  
var  
    d,k: Integer;  
    Coor: array [1..N] of PointType;  
begin  
    d := Detect;  
    InitGraph(d,k,'');  
    for k := 1 to N do with Coor[k] do  
        if odd(k) then  
            begin  
                X := 0;  
                Y := 0  
            end  
        else  
            begin  
                X := GetMaxX;  
                Y := GetMaxY  
            end;  
        DrawPoly(N,Coor);  
        ReadLn;  
        CloseGraph  
    end.
```

это значение равно 510. В этой программе ломаная задается в виде многократно накладывающихся друг на друга диагональных линий.

14.5. ДУГИ, ОКРУЖНОСТИ, ЭЛЛИПСЫ

Процедура Circle. Вычерчивает окружность. Заголовок:

Procedure Circle(*X,Y*: Integer; *R*: Word);

Здесь *X, Y* — координаты центра; *R* — радиус в пикселах.

Окружность выводится текущим цветом. Толщина линии устанавливается текущим стилем, вид линии всегда *SolidLn* (сплошная). Процедура вычерчивает правильную окружность с учетом изменения линейного размера радиуса в зависимости от его направления относительно сторон графического экрана, т.е. с учетом коэффициента *GetAspectRatio*. В связи с этим параметр *R* определяет количество пиксел в горизонтальном направлении.

В следующем примере в центре экрана создается окно, постепенно заполняющееся случайными окружностями. Для выхода из программы нажмите на любую клавишу.

Uses Graph, CRT;

var

d,r,e,x,y: Integer;

begin

{Инициализируем графику}

d := Detect;

InitGraph(*d*, *r*, '');

e := GraphResult;

if *e* <> grOK **then**

WriteLn(GraphErrorMsg(*e*))

else

begin

{Создаем окно в центре экрана}

x := GetMaxX **div** 4;

y := GetMaxY **div** 4;

Rectangle(*x,y,3*x,3*y*);

SetViewport(*x+1,y+1,3*x-1,3*y-1,ClipOn*);

{Цикл вывода случайных окружностей}

repeat

SetColor(succ(Random(white))); {Случайный цвет}

SetLineStyle(0,0,2*Random(2)+1); {и стиль линии}

x := Random(GetMaxX); {Случайное положение}

y := Random(GetMaxY); {центра окружности}

Circle(*x,y*,Random(GetMaxY **div** 4));

until KeyPressed;

if ReadKey=#0 **then** *x* := ord(ReadKey);

CloseGraph

end

end.

Процедура Arc. Чертит дугу окружности. Заголовок:

Procedure Arc(*X,Y*: Integer; BegA,EndA,*R*: Word);

Здесь X, Y — координаты центра; $BegA, EndA$ — соответственно начальный и конечный углы дуги; R — радиус.

Углы отсчитываются против часовой стрелки и указываются в градусах. Нулевой угол соответствует горизонтальному направлению вектора слева направо. Если задать значения начального угла 0 и конечного — 359, то будет выведена полная окружность. При вычерчивании дуги окружности используются те же соглашения относительно линий и радиуса, что и в процедуре *Circle*.

Вот как выглядят две дуги: одна с углами 0 и 90, вторая 270 и 540 градусов (рис.14.6):

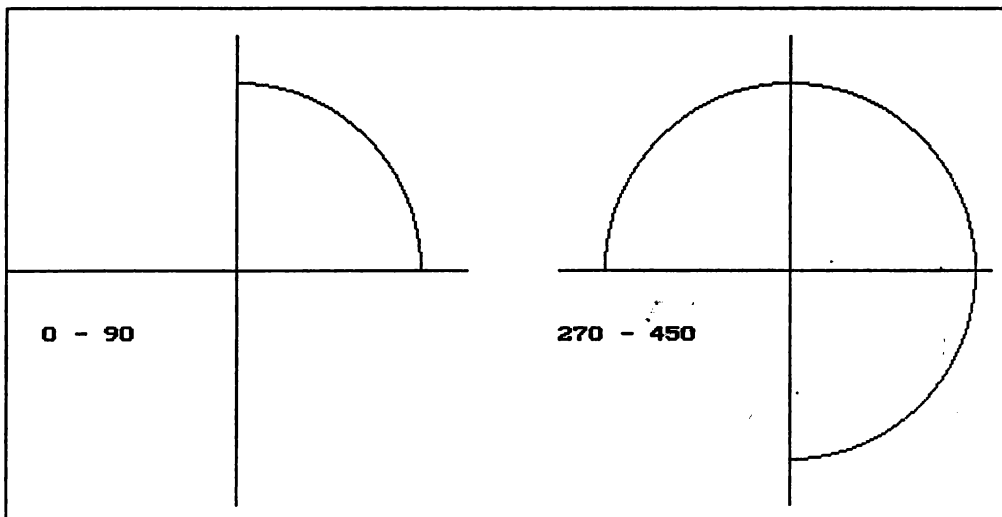


Рис.14.6. Иллюстрация процедуры Arc

Следующая программа создает это изображение:

```

Uses Graph, CRT;
var
    d, r, e : Integer;
    Xasp, Yasp: Word;
begin
    {Инициализируем графику}
    d := Detect;
    InitGraph(d, r, '');
    e := GraphResult;
    if e <> grOK then
        WriteLn(GraphErrorMsg(e))
    else
        begin
            GetAspectRatio(Xasp, Yasp);
            {R = 1/5 от вертикального размера экрана}
            r := round(Yasp*GetMaxY/5/Xasp);
            d := GetMaxX div 2;      {Смещение второго графика}

```

```

e := GetMaxY div 2;      {Положение горизонтальной оси}
{Строим левый график}
Line (0,e,5*r div 2,e);      {Горизонтальная ось}
Line (5*r div 4,e div 2,5*r div 4,3*e div 2);
Arc (5*r div 4,e,0,90,R);      {Дуга}
OutTextXY(0,e+e div 8,'0 - 90'); {Надпись}
{Правый график}
Line (d,e,d+5*r div 2,e);
Line (d+5*r div 4,e div 2, d+5*r div 4,3*e div 2);
Arc (d+5*r div 4,e,270,540,R);
OutTextXY(d,e+e div 8,'270 - 540');
{Ждем нажатия на любую клавишу}
if ReadKey=#0 then d := ord(ReadKey);
CloseGraph
end
end.

```

Процедура *GetArcCoords*. Возвращает координаты трех точек: центра, начала и конца дуги. Заголовок:

```
Procedure GetArcCoords(var Coords: ArcCoordsType);
```

Здесь *Coords* — переменная типа *ArcCoordsType*, в которой процедура возвращает координаты центра, начала и конца дуги.

Тип *ArcCoordsType* определен в модуле *Graph* следующим образом:

```

type
ArcCoordsType = record
    X,Y          : Integer; {Координаты центра}
    Xstart,Ystart: Integer; {Начало дуги}
    Xend,Yend     : Integer; {Конец дуги}
end;

```

Совместное использование процедур *Arc* и *GetArcCoords* позволяет вычерчивать сопряжения двух прямых с помощью дуг. Обратите внимание на коррекцию длины радиуса в следующем примере, в котором вычерчивается прямоугольник со скругленными углами.

```

Uses Graph,CRT;
const
    RadX = 50;      {Горизонтальный радиус}
    lx   = 400;     {Ширина}
    ly   = 100;     {Высота}
var
    d,r,e: Integer;
    coo  : ArcCoordsType;
    xl,yl: Integer;
    xa,ya: Word;
    RadY : Integer; {Вертикальный радиус}
begin

```

```

{Иницилируем графику}
d := Detect;
InitGraph(d, r, '');
e := GraphResult;
if e <> grOK then
  WriteLn(GraphErrorMsg(e))
else
  begin
    GetAspectRatio(xa, ya); {Получаем отношение сторон}
    {Вычисляем вертикальный радиус и положение фигуры
    с учетом отношения сторон экрана}
    RadY := round(RadX*(xa/ya));
    x1 := (GetMaxX-lx) div 2;
    y1 := (GetMaxY-2*RadY-ly) div 2;
    {Вычерчиваем фигуру}
    Line (x1, y1, x1+lx, y1);           {Верхняя горизонтальная}
    Arc (x1+lx, y1+RadY, 0, 90, RadX);   {Скругление}
    GetArcCoords (coo);
    with coo do
      begin
        Line (Xstart, Ystart, Xstart, Ystart+ly);
        {Правая вертикальная}
        Arc (Xstart-RadX, Ystart+ly, 270, 0, RadX);
        GetArcCoords (coo);
        Line (Xstart, Ystart, Xstart-lx, Ystart);
        {Нижняя горизонтальная}
        Arc (Xstart-lx, Ystart-RadY, 180, 270, RadX);
        GetArcCoords (coo);
        Line (Xstart, Ystart, Xstart, Ystart-ly);
        Arc (Xstart+RadX, Ystart-ly, 90, 180, RadX)
      end;
    if ReadKey=#0 then d := ord(ReadKey);
    CloseGraph
  end
end.

```

Процедура Ellipse. Вычерчивает эллипсную дугу. Заголовок:

Procedure Ellipse(X, Y: Integer; BegA, EndA, RX, RY: Word);

Здесь X, Y — координаты центра; BegA, EndA — соответственно начальный и конечный углы дуги; RX, RY — горизонтальный и вертикальный радиусы эллипса в пикселах.

При вычерчивании дуги эллипса используются те же соглашения относительно линий, что и в процедуре Circle, и те же соглашения относительно углов, что и в процедуре Arc. Если радиусы согласовать с учетом масштабного коэффициента GetAspectRatio, будет вычерчена правильная окружность.

В следующей программе вычерчиваются три эллипсных дуги (рис.14.7) при разных отношениях радиусов. Замечу, что чем выше разрешение графического экрана, тем ближе к единице отношение сторон и тем меньше первый график отличается от третьего.

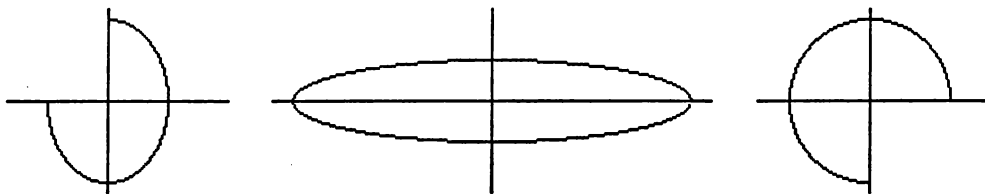


Рис.14.7. Эллипсные дуги

```

Uses Graph, CRT;
var
    d,r,e: Integer,
    xa,ya: Word;
begin
    {Инициализируем графику}
    d := Detect;
    InitGraph(d, r, '');
    e := GraphResult;
    if e <> grOK then
        WriteLn(GraphErrorMsg(e))
    else
        begin
            {Первый график}
            OutTextXY(50,40,'RX = RY');      {Надпись}
            Line (0,100,160,100);             {Ось X}
            Line (80,55,80,145);              {Ось Y}
            Ellipse (80,100,180,90,40,40);
            {Второй график}
            OutTextXY(260,40,'RX = 5*RY');
            Line (190,100,410,100);
            Line (300,55,300,145);
            Ellipse (300,100,0,359,100,20);
            {Третий график}
            OutTextXY(465,40,'Aspect Ratio');
            Line (440,100,600,100);
            Line (520,55,520,145);
            GetAspectRatio(xa, ya);
            Ellipse (520,100,0,270,40 round(40*(xa/ya)));
            if ReadKey=#0 then
                d := ord(ReadKey);
            CloseGraph
        end
    end.

```

14.6. КРАСКИ, ПАЛИТРЫ, ЗАПОЛНЕНИЯ

Процедура SetColor. Устанавливает текущий цвет для выводимых линий и символов. Заголовок:

Procedure SetColor(Color: Word);

Здесь *Color* — текущий цвет.

В модуле *Graph* определены точно такие же константы для задания цвета, как и в модуле *CRT* (см. п.13.2).

Функция GetColor. Возвращает значение типа *Word*, содержащее код текущего цвета. Заголовок:

Function GetColor: Word;

Функция GetMaxColor. Возвращает значение типа *Word*, содержащее максимальный доступный код цвета, который можно использовать для обращения к *SetColor*. Заголовок:

Function GetMaxColor: Word;

Процедура SetBkColor. Устанавливает цвет фона. Заголовок:

Procedure SetBkColor(Color: Word);

Здесь *Color* — цвет фона.

В отличие от текстового режима, в котором цвет фона может быть только темного оттенка, в графическом режиме он может быть любым. Установка нового цвета фона немедленно изменяет цвет графического экрана. Это означает, что нельзя создать изображение, два участка которого имели бы разный цвет фона. Для CGA-адаптера в режиме высокого разрешения установка цвета фона изменяет цвет активных пиксел. Замечу, что после замены цвета фона на любой, отличный от 0 (*Black*) цвет, Вы не сможете более использовать цвет 0 как черный, он будет заменяться на цвет фона, т.к. процедуры модуля *Graph* интерпретируют цвет с номером 0 как цвет фона. Это означает, в частности, что Вы уже не сможете вернуть фону черный цвет!

Если Ваш ПК оснащен цветным экраном, следующая программа продемонстрирует работу процедуры *SetBkColor*. Программа выводит десять вложенных друг в друга прямоугольников, после чего циклически меняет цвет фона. Для выхода из программы достаточно нажать на любую клавишу.

Uses Graph, CRT;

const

```
NC: array [0..15] of String [12] =  
  ('Black', 'Blue', 'Green', 'Cyan', 'Red', 'Magenta',  
  'Brown', 'LightGray', 'DarkGray', 'LightBlue',  
  'LightGreen', 'LightCyan', 'LightRed',  
  'LightMagenta', 'Yellow', 'White');
```

```

var
  d, r, e, k, color, dx, dy: Integer;
begin
  {Инициализируем графику}
  d := Detect;
  InitGraph(d, r, '');
  e := GraphResult;
  if e <> grOK then
    WriteLn(GraphErrorMsg(e))
  else
    begin
      {Выводим текст в центре экрана}
      OutTextXY(200, GetMaxY div 2, 'BACKGROUND COLOR');
      dx := GetMaxX div 30;  {Приращение длины}
      dy := GetMaxY div 25;  {Приращение высоты}
      for k := 0 to 9 do      {Выводим 10 прямоугольников}
        Rectangle(k*dx, k*dy, GetMaxX-k*dx, GetMaxY-k*dy);
        color := black;       {Начальный цвет фона}
      repeat                  {Цикл смены фона}
        SetBkColor(color);
        SetFillStyle(0, Color);
        Bar(345, GetMaxY div 2, 440, GetMaxY div 2+8);
        OutTextXY(345, GetMaxY div 2, NC[color]);
        delay(1000);
        inc(color);
        if color > White then
          color := Black
        until KeyPressed;
        if ReadKey=#0 then k := ord(ReadKey);
      CloseGraph
    end
  end.

```

Функция GetBkColor. Возвращает значение типа *Word*, содержащее текущий цвет фона. Заголовок:

Function GetBkColor: Word;

Процедура SetPalette. Заменяет один из цветов палитры на новый цвет. Заголовок:

Procedure SetPalette(N: Word; Color: ShortInt);

Здесь *N* — номер цвета в палитре; *Color* — номер вновь устанавливаемого цвета.

Данная процедура может работать только с адаптерами *EGA* или *VGA*. Она не должна использоваться с *IBM8514* или 256-цветным вариантом *VGA* — для этих адаптеров предназначена особая процедура *SetRGBPalette* (см. ниже). Первоначальное размещение цветов в палитрах *EGA/VGA*

соответствует последовательности их описания константами *Black,...,White*, т.е. цвет с индексом 0 — черный, 1 — синий, 2 — зеленый и т.д. После обращения к процедуре все фрагменты изображения, выполненные цветом с индексом *N* из палитры цветов, получают цвет *Color*. Например, если выполнить оператор

```
SetPalette(2,White);
```

то цвет с индексом 2 (первоначально это — бирюзовый цвет *Cyan*) будет заменен на белый. Замечу, что цвет с индексом 0 отождествляется с цветом фона и может изменяться наряду с любым другим цветом.

Следующая программа выводит на экран ряд прямых разного цвета и затем случайным образом меняет цвета палитры.

```
Uses Graph, CRT;
```

```
var
```

```
  d,r,e,N,k,color: Integer;
```

```
  Palette          : PaletteType;
```

```
begin
```

```
  {Иницилируем графику}
```

```
  d := Detect;
```

```
  InitGraph(d, r, '');
```

```
  e := GraphResult;
```

```
  if e <> grOK then
```

```
    WriteLn(GraphErrorMsg(e))
```

```
  else
```

```
    begin
```

```
      {Выбираем толстые сплошные линии}
```

```
      SetLineStyle(SolidLn, 0, ThickWidth);
```

```
      GetPalette(Palette);           {Текущая палитра}
```

```
      for Color := 0 to Palette.Size-1 do
```

```
        begin
```

```
          SetColor(Color);
```

```
          Line(GetMaxX div 3,Color*10,2*GetMaxX div 3,Color*10)
```

```
        end;
```

```
      {Меняем палитру и ждем инициативы пользователя}
```

```
      while not KeyPressed do
```

```
        for e := 0 to Palette.Size-1 do
```

```
          SetPalette(e,Random(Palette.Size));
```

```
        if ReadKey=#0 then d := ord(ReadKey);
```

```
        CloseGraph
```

```
      end
```

```
end.
```

Процедура GetPalette. Возвращает размер и цвета текущей палитры.
Заголовок:

```
Procedure GetPalette(var PaletteInfo: PaletteType);
```

Здесь *PaletteInfo* — переменная типа *PaletteType*, возвращающая размер и цвета палитры.

В модуле *Graph* определена константа

```
const
    MaxColors = 15;
```

и тип

```
type
    PaletteType = record
        Size : Word; {Количество цветов в палитре}
        Colors : array [0..MaxColors] of ShortInt
            {Номера входящих в палитру цветов}
    end;
```

С помощью следующей программы можно вывести на экран номера всех возможных цветов из текущей палитры.

```
Uses Graph;
var
    Palette: PaletteType;
    d,r,e,k: Integer;
begin
    {Инициализируем графику}
    d := Detect;
    InitGraph(d, r, '');
    e := GraphResult;
    if e <> grOk then
        WriteLn(GraphErrorMsg(e))
    else
        begin
            GetPalette(Palette);    {Получаем палитру}
            CloseGraph; {Возвращаемся в текстовый режим}
            with Palette do
                {Выводим номера цветов}
                for k := 0 to pred(Size) do
                    Write(Colors[k]:5);
                end
            end
        end
    end.
```

Процедура SetAllPalette. Изменяет одновременно несколько цветов палитры. Заголовок процедуры:

```
Procedure SetAllPalette(var Palette);
```

Параметр *Palette* в заголовке процедуры описан как нетипизированный параметр. Первый байт этого параметра должен содержать длину *N* палитры, остальные *N* байты — номера вновь устанавливаемых цветов в диапазоне от -1 до *MaxColors*. Код -1 означает, что соответствующий цвет исходной палитры не меняется.

В следующей программе происходит одновременная смена сразу всех цветов палитры.

```

Uses Graph, CRT;
var
  Palette: array [0..MaxColors] of ShortInt;
  d,r,e,k: Integer;
begin
  {Иницилируем графику}
  d := Detect;
  InitGraph(d, r, '');
  e := GraphResult;
  if e <> grOk then
    WriteLn(GraphErrorMsg(e))
  else
    begin
      {Выбираем толстые сплошные линии}
      SetLineStyle(SolidLn, 0, ThickWidth);
      {Выводим линии всеми доступными цветами}
      for k := 1 to GetMaxColor do
        begin
          SetColor(k);
          Line(GetMaxX div 3, k*10, 2*GetMaxX div 3, k*10)
        end;
      Palette[0] := MaxColors;           {Размер палитры}
      repeat                             {Цикл смены палитры}
        for k := 1 to MaxColors do
          Palette[k] := Random(succ(MaxColors));
          SetAllPalette(Palette)
        until KeyPressed;
        if ReadKey=#0 then k := ord(ReadKey);
      CloseGraph
    end
  end.

```

Функция *GetPaletteSize*. Возвращает значение типа *Integer*, содержащее размер палитры (максимальное количество доступных цветов).
Заголовок:

```
Function GetPaletteSize: Integer;
```

Процедура *GetDefaultPalette*. Возвращает структуру палитры, установленную по умолчанию (в режиме автонастройки). Заголовок:

```
Procedure GetDefaultPalette(var Palette: PaletteType);
```

Здесь *Palette* — переменная типа *PaletteType* (см. процедуру *GetPalette*), в которой возвращаются размер и цвета палитры.

Процедура SetFillStyle. Устанавливает стиль (тип и цвет) заполнения. Заголовок:

Procedure SetFillStyle(Fill,Color: Word);

Здесь *Fill* — тип заполнения; *Color* — цвет заполнения.

С помощью заполнения можно покрывать какие-либо фрагменты изображения периодически повторяющимся узором. Для указания типа заполнения используются следующие предварительно определенные константы:

const

```
EmptyFill      = 0;    {Заполнение фоном (узор отсутствует)}
SolidFill      = 1;    {Сплошное заполнение}
LineFill       = 2;    {Заполнение -----}
LtSlashFill    = 3;    {Заполнение ///////}
SlashFill      = 4;    {Заполнение утолщенными ///}
BkSlashFill    = 5;    {Заполнение утолщенными \\\}
LtBkSlashFill  = 6;    {Заполнение \\\\\\\}
HatchFill      = 7;    {Заполнение +++++++}
XHatchFill     = 8;    {Заполнение xxxxxxxx}
InterleaveFill = 9;    {Заполнение прямоугольную клеточку}
WideDotFill    = 10;   {Заполнение редкими точками}
CloseDotFill   = 11;   {Заполнение частыми точками}
UserFill       = 12;   {Узор определяется пользователем}
```

Программа из следующего примера продемонстрирует Вам все стандартные типы заполнения.

Uses Graph, CRT;

var

d,r,e,k,j,x,y: Integer;

begin

{Инициализируем графику}

d := Detect;

InitGraph(d, r, '');

e := GraphResult;

if e <> grOk **then**

WriteLn(GraphErrorMsg(e))

else

begin

x := GetMaxX div 6; {Положение графика}

y := GetMaxY div 5; {на экране}

for j := 0 **to** 2 **do** {Два ряда}

for k := 0 **to** 3 **do** {По четыре квадрата}

begin

Rectangle((k+1)*x, (j+1)*y, (k+2)*x, (j+2)*y);

SetFillStyle(k+j*4,j+1);

Bar((k+1)*x+1, (j+1)*y+1, (k+2)*x-1, (j+2)*y-1)

end;

```

    if ReadKey=#0 then k := ord(ReadKey);
    CloseGraph
  end
end.

```

Если параметр *Fill* имеет значение 12 (*UserFill*), то рисунок узора определяется программистом путем обращения к процедуре *SetFillPattern*.

Процедура SetFillPattern. Устанавливает образец рисунка и цвет штриховки. Заголовок:

```

Procedure SetFillPattern(Pattern: FillPatternType;
                          Color: Word);

```

Здесь *Pattern* — выражение типа *FillPatternType*; устанавливает образец рисунка для *Fill* = *UserFill* в процедуре *SetFillStyle*; *Color* — цвет заполнения.

Образец рисунка задается в виде матрицы из 8x8 пиксел и может быть представлен массивом из 8 байт следующего типа:

```

type
  FillPatternType = array [1..8] of Byte;

```

Каждый разряд любого из этих байтов управляет светимостью пиксела, причем первый байт определяет 8 пиксел первой строки на экране, второй байт — 8 пиксел второй строки и т.д.

На рис.14.8 показан пример двух образцов заполнения. На рисунке черточкой обозначается несветящийся пиксел, а прямоугольником — светящийся. Для каждого 8 пиксел приводится шестнадцатиричный код соответствующего байта.

Следующая программа заполняет этими образцами две прямоугольных области экрана.

Образец	Значение байта	Образец	Значение байта
- ■ - - ■ - - ■	\$49	- - - - - - - -	\$00
■ - - ■ - - ■ -	\$92	- - - ■ - - - -	\$18
- ■ - - ■ - - ■	\$49	- - ■ - - - - -	\$24
■ - - ■ - - ■ -	\$92	- ■ - - - - - ■	\$42
- ■ - - ■ - - ■	\$49	- - - - - - - ■	\$42
■ - - ■ - - ■ -	\$92	- - ■ - - - - -	\$24
- ■ - - ■ - - ■	\$49	- - - ■ - - - -	\$18
■ - - ■ - - ■ -	\$92	- - - - - - - -	\$00

Рис.14.8. Образцы заполнения и их коды

```

Uses Graph, CRT;

```

```

const

```

```

  patt1: FillPatternType= ($49,$92,$49,$92,$49,$92,$49,$92);
  patt2: FillPatternType= ($00,$18,$24,$42,$42,$24,$18,$00);

```

```

var
  d,r,e: Integer;
begin
  {Инициализируем графику}
  d := Detect;
  InitGraph(d, r, '');
  e := GraphResult;
  if e <> grOk then
    WriteLn(GraphErrorMsg(e))
  else
    begin
      if d=CGA then
        SetGraphMode(0);           {Устанавливаем цвет для CGA}
        SetFillStyle(UserFill,White);
        {Левый верхний квадрат}
        SetFillPattern(Patt1,1);
        Bar(0,0,GetMaxX div 2, GetMaxY div 2);
        {Правый нижний квадрат}
        SetFillPattern(Patt2,2);
        Bar(GetMaxX div 2,GetMaxY div 2,GetMaxX,GetMaxY);
        if ReadKey=#0 then d := ord(ReadKey);
        CloseGraph
      end
    end.
end.

```

Если при обращении к процедуре указан недопустимый код цвета, вызов процедуры игнорируется и сохраняется ранее установленный образец заполнения. В частности, если в предыдущем примере убрать оператор

```

if d=CGA then
  SetGraphMode(0);

```

устанавливающий цветной режим работы CGA-адаптера, на экран ПК, оснащенного адаптером этого типа, будут выведены два одинаковых прямоугольника, так как обращение

```
SetFillPattern(patt2, 2);
```

содержит недопустимо большой для данного режима код цвета и обращение игнорируется. Сказанное, однако, не относится к процедуре *SetFillStyle* для значения параметра *Fill* в диапазоне от 0 до 11: программа будет нормально работать и в режиме высокого разрешения CGA-адаптера, причем все цвета палитры, кроме цвета фона, при этом замечаются на белый.

Процедура *GetFillPattern*. Возвращает образец заполнения, установленный ранее процедурой *SetFillPattern*. Заголовок:

```
Procedure GetFillPattern(var Pattern: FillPatternType);
```

Здесь *Pattern* — переменная типа *FillPatternType*, в которой возвращается образец заполнения.

Если программа не устанавливала образец с помощью процедуры *SetFillPattern*, массив *Pattern* заполняется байтами со значением 255 (\$FF).

Процедура *GetFillSettings*. Возвращает текущий стиль заполнения.
Заголовок:

```
Procedure GetFillSettings(var PattInfo: FillSettingsType);
```

Здесь *PattInfo* — переменная типа *FillSettingsType*, в которой возвращается текущий стиль заполнения.

В модуле *Graph* определен тип:

```
type  
  FillSettingsType = record  
    Pattern: Word; {Образец}  
    Color : Word   {Цвет}  
  end;
```

Поля *Pattern* и *Color* в этой записи имеют то же назначение, что и аналогичные параметры при обращении к процедуре *SetFillStyle*.

Процедура *SetRGBPalette*. Устанавливает цветовую гамму при работе с дисплеем *IBM 8514* и адаптером *VGA*. Заголовок:

```
Procedure SetRGBPalette(ColNum, RedVal, GreenVal, BlueVal:  
  Integer);
```

Здесь *ColNum* — номер цвета; *RedVal*, *GreenVal*, *BlueVal* — выражения типа *Integer*, устанавливающие интенсивность соответственно красной, зеленой и синей составляющих цвета.

Эта процедура может работать только с дисплеем *IBM 8514*, а также с адаптером *VGA*, использующим видеопамять объемом 256 Кбайт. В первом случае параметр *ColNum* задается числом в диапазоне 0...255, во втором — в диапазоне 0...15. Для установки интенсивности используются 6 старших разрядов младшего байта любого из параметров *RedVal*, *GreenVal*, *BlueVal*.

В следующей программе в центре экрана выводится прямоугольник белым цветом, после чего этот цвет случайно изменяется с помощью процедуры *SetRGBPalette*. Для выхода из программы нужно нажать любую клавишу.

```
Uses Graph, CRT;  
var  
  Driver, Mode, Err, x1, y1: Integer;  
begin  
  {Инициализируем графический режим}  
  Driver := Detect;  
  InitGraph(Driver, Mode, '');  
  Err := GraphResult;  
  if Err <> 0 then  
    WriteLn(GraphErrorMsg(Err))
```

```

else if Driver in [IBM8514, VGA] then
  begin
    {Выводим прямоугольник в центре экрана}
    x1 := GetMaxX div 4;
    y1 := GetMaxY div 4;
    SetColor(15);
    Bar(x1,y1,3*x1,3*y1);
    {Изменяем белый цвет на случайный}
    while not KeyPressed do
      SetRGBPalette(15,Random(256),Random(256),Random(256));
    CloseGraph
  end
else
  begin
    CloseGraph;
    WriteLn('Адаптер не поддерживает ',
      'RGB-режим управления цветами')
  end
end.

```

Процедура FloodFill. Заполняет произвольную замкнутую фигуру, используя текущий стиль заполнения (узор и цвет). Заголовок:

Procedure FloodFill(X,Y: Integer; Border: Word);

Здесь X, Y — координаты любой точки внутри замкнутой фигуры; Border — цвет граничной линии.

Если фигура незамкнута, заполнение «разольется» по всему экрану.

Следует учесть, что реализованный в процедуре алгоритм просмотра границ замкнутой фигуры не отличается совершенством. В частности, если выводятся подряд две пустые строки, заполнение прекращается. Такая ситуация обычно возникает при заполнении небольших фигур с использованием типа *LtSlashFill*. В фирменном руководстве по Турбо Паскалю рекомендуется, по возможности, вместо процедуры *FloodFill* использовать *FillPoly* (заполнение прямоугольника).

Следующая программа демонстрирует заполнение случайных окружностей. Сначала в центре экрана создается окно, в котором заполняется небольшой прямоугольник. Часть прямоугольника останется незаполненной, в чем Вы можете убедиться, так как программа в этот момент приостанавливает работу, ожидая нажатия на клавишу *Enter*. Затем осуществляется вывод и заполнение случайных окружностей до тех пор, пока не будет нажата любая клавиша. Замечу, что прямоугольник заполняется полностью, если вместо типа *LtSlashFill* (косая штриховка линиями обычной толщины) используется *SlashFill* (штриховка утолщенными линиями). Если программа будет работать достаточно долго, она может «зависнуть», что лишний раз свидетельствует о несовершенстве реализованного в ней алгоритма.

Uses Graph, CRT;


```

var
  d, r, e, x, y, c : Integer;
begin
  {Инициализируем графику}
  d := Detect;
  InitGraph(d, r, '');
  e := GraphResult;
  if e <> grOk then
    WriteLn(GraphErrorMsg(e))
  else
    begin
      {Создаем прямоугольное окно}
      x := GetMaxX div 4;
      y := GetMaxY div 4;
      Rectangle(x, y, 3*x, 3*y);
      SetViewPort(x+1, y+1, 3*x-1, 3*y-1, ClipOn);
      {Демонстрируем заливку маленького прямоугольника}
      SetFillStyle(LtSlashFill, GetMaxColor);
      Rectangle(0, 0, 8, 20);
      FloodFill(1, 1, GetMaxColor);
      OutTextXY(10, 25, 'Press Enter...');
      ReadLn;
      {Ждем нажатия Enter}
      {Выводим окружности до тех пор, пока
      не будет нажата любая клавиша}
      repeat
        {Определяем случайный стиль заливки}
        SetFillStyle(Random(12), Random(GetMaxColor+1));
        {Задаем координаты центра и цвет окружности}
        x := Random (GetMaxX div 2);
        y := Random (GetMaxY div 2);
        c := Random (succ(GetMaxColor));
        SetColor(c);
        {Выводим и заливаем окружность}
        Circle(x, y, Random(GetMaxY div 5));
        FloodFill (x, y, c)
      until KeyPressed;
      if ReadKey=#0 then x := ord(ReadKey);
      CloseGraph
    end
  end.
end.

```

Процедура Bar. Заполняет прямоугольную область экрана. Заголовок:

Procedure Bar (X1, Y1, X2, Y2: Integer);

Здесь X1...Y2 — координаты левого верхнего (X1, Y1) и правого нижнего (X2, Y2) углов закрашиваемой области.

Процедура закрашивает (но не обводит) прямоугольник текущим образом узора и текущим цветом, которые устанавливаются процедурой *SetFillStyle*.

Следующая программа дает красивые цветовые эффекты (закраска случайных прямоугольников).

```
Uses Graph, CRT;
var
  d, r, e : Integer;
begin
  {Инициализируем графику}
  d := Detect;
  InitGraph(d, r, '');
  e := GraphResult;
  if e <> grOk then
    WriteLn(GraphErrorMsg(e))
  else
    begin
      {Создаем окно в центре экран}
      d := GetMaxX div 4;
      r := GetMaxY div 4;
      Rectangle(d, r, 3*d, 3*r);
      SetViewport(d+1, r+1, 3*d-1, 3*r-1, ClipOn);
      {Цикл вывода и закрашки случайных прямоугольников}
      repeat
        SetFillStyle(Random(12), Random(succ(GetMaxColor)));
        Bar(Random(GetMaxX), Random(GetMaxY),
            Random(GetMaxX), Random(GetMaxY));
      until KeyPressed;
      if ReadKey=#0 then d := ord(ReadKey);
      CloseGraph
    end
  end.
end.
```

Процедура Bar3D. Вычерчивает трехмерное изображение параллелепипеда и закрашивает его переднюю грань. Заголовок:

Procedure Bar3D (X1,Y1,X2,Y2,Depth: Integer; Top: Boolean);

Здесь X1...Y2 — координаты левого верхнего (X1, Y1) и правого нижнего (X2, Y2) углов передней грани; Depth — третье измерение трехмерного изображения («глубина») в пикселах; Top — способ изображения верхней грани.

Если параметр Top имеет значение True, верхняя грань параллелепипеда вычерчивается, в противном случае — не вычерчивается (этот вариант используется для изображения поставленных друг на друга параллелепипедов, см. следующий пример). В качестве значения этого параметра может использоваться одна из следующих констант, определенных в модуле Graph:

```

const
    TopOn  = True;
    TopOff = False;

```

При вычерчивании используется текущий стиль линий (*SetLineStyle*) и текущий цвет (*SetColor*). Передняя грань заливается текущим стилем за-полнения (*SetFillStyle*).

Процедура обычно применяется при построении столбиковых диаграмм. Следует учесть, что параллелепипед «прозрачен», т.е. за его незакра-шенными гранями могут быть видны другие элементы изображения.

Следующая программа иллюстрирует различные аспекты применения процедуры *Bar3D*.

```

Uses Graph, CRT;
var
    d, r, e: Integer;
begin
    {Иницилируем графику}
    d := Detect;
    InitGraph(d, r, '');
    e := GraphResult;
    if e <> grOk then
        WriteLn(GraphErrorMsg(e))
    else
        begin
            {Столбик с верхней гранью:}
            Bar3D (80, 100, 120, 180, 15, TopOn);
            {Столбик без верхней грани:}
            Bar3D (150, 150, 190, 180, 15, TopOff);
            {Этот столбик "стоит" на следующем и прозрачен:}
            Bar3D (230, 50, 250, 150, 15, TopOn);
            Bar3D (220, 150, 260, 180, 15, TopOn);
            {У этого столбика нет верхней грани, и поэтому он
             не мешает поставленному на него сверху:}
            Bar3D (300, 150, 340, 180, 15, TopOff);
            SetLineStyle(3,0,1);
            SetColor(Yellow);
            SetFillStyle(LtSlashFill,Yellow);
            Bar3D (300, 50, 340, 150, 15, TopOn);
            if ReadKey=#0 then d := ord(ReadKey);
            CloseGraph;
        end
    end.

```

Процедура FillPoly. Обводит линией и закрашивает замкнутый многоугольник. Заголовок:

```

Procedure FillPoly(N: Word; var Coords);

```

Здесь N — количество вершин замкнутого многоугольника; *Coords* — переменная типа *PointType*, содержащая координаты вершин.

Координаты вершин задаются парой значений типа *Integer*: первое определяет горизонтальную, второе — вертикальную координаты. Для них можно использовать следующий определенный в модуле тип:

```

type
  PointType = record
    x, y : Integer
  end;

```

Стиль и цвет линии контура задаются процедурами *SetLineStyle* и *SetColor*, тип и цвет заливки — процедурой *SetFillStyle*.

В следующем примере на экран выводятся случайные закрашенные многоугольники.

```

Uses Graph, CRT;
var
  d, r, e: Integer;
  p      : array [1..6] of PointType;
  n, k   : Word;
begin
  {Иницилируем графику}
  d := Detect;
  InitGraph(d, r, '');
  e := GraphResult;
  if e <> grOk then
    WriteLn(GraphErrorMsg(e))
  else
    begin
      {Создаем окно в центре экрана}
      d := GetMaxX div 4;
      r := GetMaxY div 4;
      Rectangle(d, r, 3*d, 3*r);
      SetViewport(d+1, r+1, 3*d-1, 3*r-1, ClipOn);
      {Цикл вывода случайных закрашенных многоугольников}
      repeat
        {Выбираем случайный цвет и узор}
        SetFillStyle(Random(12), Random(succ(GetMaxColor)));
        SetColor (Random(succ(GetMaxColor)));
        {Назначаем случайные координаты}
        n := Random (4) + 3;
        for k := 1 to n do with p[k] do
          begin
            x := Random (GetMaxX div 2);
            y := Random (GetMaxY div 2)
          end;
        FillPoly (n, p)
      until KeyPressed;
    end
  end

```

{Выводим и закрашиваем}

```

    if ReadKey=#0 then k := ord(ReadKey);
    CloseGraph
  end
end.

```

Процедура FillEllipse. Обводит линией и заполняет эллипс. Заголовок:

```

Procedure FillEllipse(X,Y,RX,RY: Integer);

```

Здесь X , Y — координаты центра; RX , RY — горизонтальный и вертикальный радиусы эллипса в пикселах.

Эллипс обводится линией, заданной процедурами *SetLineStyle* и *SetColor*, и заполняется с использованием параметров, установленных процедурой *SetFillStyle*.

Процедура Sector. Вычерчивает и заполняет эллипсный сектор. Заголовок:

```

Procedure Sector(X,Y: Integer; BegA,EndA,RX,RY: Word);

```

Здесь *BegA*, *EndA* — соответственно начальный и конечный углы эллипсного сектора. Остальные параметры обращения аналогичны параметрам процедуры *FillEllipse*.

В следующей программе на экран выводятся случайные закрашенные эллипсы и секторы. Для выхода из программы нажмите любую клавишу.

```

Uses Graph, CRT;

```

```

var
  d, r, e : Integer;
begin
  {Инициализируем графику}
  d := Detect;
  InitGraph(d, r, '');
  e := GraphResult;
  if e <> grOk then
    WriteLn(GraphErrorMsg(e))
  else
    begin
      {Создаем окно в центре экрана}
      d := GetMaxX div 4;
      r := GetMaxY div 4;
      Rectangle(d,r,3*d,3*r);
      SetViewport(d+1,r+1,3*d-1,3*r-1,ClipOn);
      {Цикл вывода}
      repeat
        SetFillStyle(Random(12),
          Random(succ(GetMaxColor)));
        SetColor(Random(succ(GetMaxColor)));
        Sector(Random(GetMaxX div 2),Random(GetMaxY div 2),
          Random(360),Random(360),Random(GetMaxX div 5),

```

```

Random(GetMaxY div 5));
FillEllipse(Random(GetMaxX div 2),
            Random(GetMaxY div 2), Random(GetMaxX div 5),
            Random(GetMaxY div 5))
until KeyPressed;
if ReadKey=#0 then d := ord(ReadKey);
CloseGraph
end
end.

```

Процедура PieSlice. Вычерчивает и заполняет сектор окружности.
Заголовок:

```
Procedure PieSlice(X,Y: Integer; BegA,EndA,R: Word);
```

В отличие от процедуры *Sector*, указывается лишь один горизонтальный радиус *R*, остальные параметры аналогичны параметрам процедуры *Sector*.

Сектор обводится линией, заданной процедурами *SetLineStyle* и *SetColor*, и заполняется с помощью параметров, определенных процедурой *SetFillStyle*. Процедуру удобно использовать при построении круговых диаграмм, как, например, в следующей программе (рис.14.9).

```

Uses Graph, CRT;
var
    d, r, e : Integer;
begin
    {Иницилируем графический режим}
    d := Detect;

```

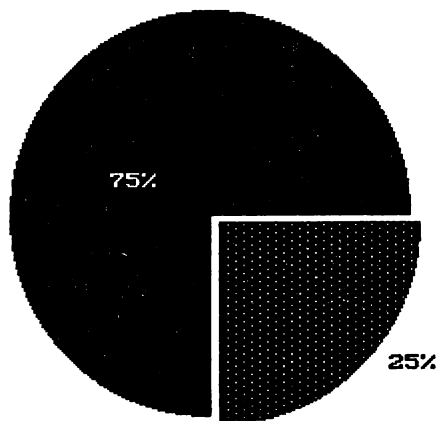


Рис.14.9. Иллюстрация процедуры PieSlice

```

InitGraph(d, r, '');
e := GraphResult;
if e <> grOk then
    WriteLn(GraphErrorMsg(e))

```

```

else
  begin
    {Выводим маленький сектор}
    SetFillStyle(WideDotFill, White);
    PieSlice(GetMaxX div 2+5, GetMaxY div 2+4, 270, 360, 100);
    {Выводим большой сектор}
    SetFillStyle (SolidFill, Red);
    PieSlice (GetMaxX div 2, GetMaxY div 2, 0, 270, 100);
    {Выводим надписи}
    OutTextXY (GetMaxX div 2+90, GetMaxY div 2+70, '25%');
    OutTextXY(GetMaxX div 2-50, GetMaxY div 2-20, '75%');
    {Ждем нажатия на любую клавишу}
    if ReadKey=#0 then d := ord(ReadKey);
    CloseGraph
  end
end.

```

14.7. СОХРАНЕНИЕ И ВЫДАЧА ИЗОБРАЖЕНИЙ

Функция ImageSize. Возвращает размер памяти в байтах, необходимый для размещения прямоугольного фрагмента изображения. Заголовок:

Function ImageSize(X1,Y1,X2,Y2: Integer): Word;

Здесь $X1...Y2$ — координаты левого верхнего ($X1, Y1$) и правого нижнего ($X2, Y2$) углов фрагмента изображения.

Процедура GetImage. Помещает в память копию прямоугольного фрагмента изображения. Заголовок:

Procedure GetImage(X1,Y1,X2,Y2: Integer; var Buf)

Здесь $X1...Y2$ — координаты углов фрагмента изображения; *Buf* — переменная или участок кучи, куда будет помещена копия видеопамати с фрагментом изображения.

Размер *Buf* должен быть не меньше значения, возвращаемого функцией *ImageSize* с теми же координатами $X1...Y2$.

Процедура PutImage. Выводит в заданное место экрана копию фрагмента изображения, ранее помещенную в память процедурой *GetImage*. Заголовок:

Procedure PutImage(X,Y: Integer; var Buf; Mode: Word);

Здесь X, Y — координаты левого верхнего угла того места на экране, куда будет скопирован фрагмент изображения; *Buf* — переменная или участок кучи, откуда берется изображение; *Mode* — способ копирования.

Как видим, координаты правого нижнего угла не указываются, так как они полностью определяются размерами вновь выводимой на экран копии изображения. Координаты левого верхнего угла могут быть какими угодно, лишь бы только выводимая копия уместилась в пределах экрана (если

копия не может разместиться на экране, она не выводится и экран остается без изменений).

Параметр *Mode* определяет способ взаимодействия вновь размещаемой копии с уже имеющимся на экране изображением. Взаимодействие осуществляется путем применения кодируемых этим параметром логических операций к каждому биту копии и изображения. Для указания применяемой логической операции можно использовать одну из следующих предварительно определенных констант:

const

```
NormalPut= 0; {Замена существующего изображения на копию}
XorPut   = 1; {Исключительное ИЛИ}
OrPut    = 2; {Объединительное ИЛИ}
AndPut   = 3; {Логическое И}
NotPut   = 4; {Инверсия изображения}
```

Наиболее часто используются операции *NormalPut*, *XORPut* и *NotPut*. Первая из них просто стирает часть экрана и на это место помещает копию из памяти в том виде, как она там сохраняется. Операция *NotPut* делает то же самое, но копия выводится в инверсном виде. Для монохромного режима это означает замену светящихся пиксел на темные и наоборот. В цветном режиме операция *NotPut* применяется к коду цвета каждого пиксела. Например, для *White* (код 15 или в двоичном виде 1111) эта операция даст код 0000 = 0 = *Black*, для *Red* = 4 = 0100 получим 1011 = 11 = *LightCyan* и т.д. Операция *XORPut*, примененная к тому же месту экрана, откуда была получена копия, сотрет эту часть экрана. Если операцию применить дважды к одному и тому же участку, вид изображения на экране не изменится. Таким способом можно довольно просто перемещать изображения по экрану, создавая иллюзию движения.

Следующая программа рисует «Неопознанный Летающий Объект» — летающую тарелку на звездном фоне (рис.14.10).

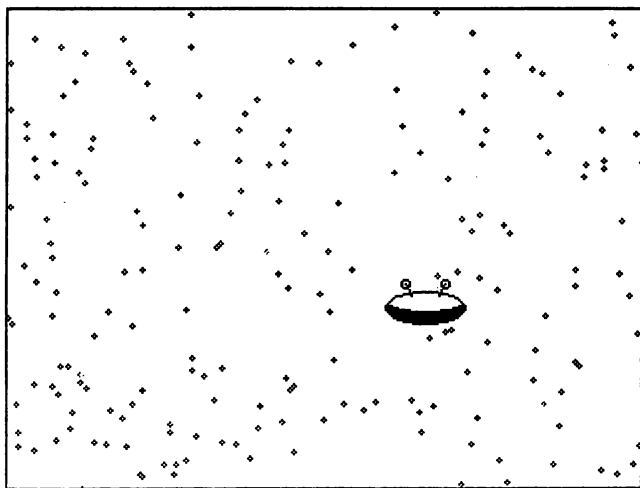


Рис.14.10. Иллюстрация процедур *GetImage/PutImage*

Uses Graph, CRT;

const

 r = 20; {Характерный размер НЛО}
 pause = 50; {Длительность паузы}

var

 d,m,e,xm,ym,x,y,lx,ly,rx,ry,
 Size,i,dx,dy,Width,Height: Integer;
 Saucer : Pointer;

label

 loop;

begin

 {Иницилируем графику}

 d := Detect;

 InitGraph(d, m, '');

 e := GraphResult;

if e <> grOk **then**

 WriteLn(GraphErrorMsg(e))

else

begin

 x := r*5;

 y := r*2;

 xm := GetMaxX **div** 4;

 ym := GetMaxY **div** 4;

 {Создаем "тарелку" из двух эллипсов с усами антенн}

 Ellipse (x,y,0,360,r,r **div** 3+2);

 Ellipse (x,y-4,190,357,r,r **div** 3);

 Line (x+7,y-6,x+10,y-12);

 Line (x-7,y-6, x-10, y-12);

 Circle (x+10,y-12,2);

 Circle (x-10,y-12,2);

 FloodFill(x+1,y+4,White);

 {Определяем габариты НЛО и помещаем его в кучу}

 lx := x-r-1;

 ly := y-14;

 rx := x+r+1;

 ry := y+r **div** 3+3;

 Width := rx - lx + 1;

 Height:= ry - ly + 1;

 Size := ImageSize(lx, ly, rx, ry);

 GetMem (Saucer, Size);

 GetImage (lx, ly, rx, ry, Saucer^);

 {Стираем построенное}

 PutImage (lx, ly, Saucer^, XorPut);

 {Создаем звездное небо}

 Rectangle (xm,ym,3*xm,3*ym);

 SetViewport (xm+1,ym+1,3*xm-1,3*ym-1,ClipOn);

 xm := 2*xm;

 ym := 2*ym;

```

for i:=1 to 200 do
  PutPixel (Random(xm), Random(ym), White);
  {Задаем начальное положение НЛО и направление движения}
  x := xm div 2;
  y := ym div 2;
  dx := 10;
  dy := 10;
  {Основной цикл}
  repeat
    PutImage(x,y,Saucer^,XorPut);      {Изображаем НЛО на}
    Delay(pause);                      {новом месте и после}
    PutImage (x, y, Saucer^, XorPut); {паузы стираем его}
    {Получаем новые координаты}
loop:  x := x+dx;
      y := y+dy;
      {НЛО достиг границы экрана?}
      if (x<0) or (x+Width+1>xm) or
        (y<0) or (y+Height+1>ym) then
        begin {Да - НЛО достиг границы: меняем
              направление его перемещения}
          x := x-dx; := y-dy;
          dx := GetMaxX div 10 - Random(GetMaxX div 5);
          dy := GetMaxY div 30 - Random(GetMaxY div 15);
          goto loop
        end
      until KeyPressed;
      if ReadKey=#0 then x := ord(ReadKey);
      CloseGraph
    end
end.

```

14.8. ВЫВОД ТЕКСТА

Описываемые ниже стандартные процедуры и функции поддерживают вывод текстовых сообщений в графическом режиме. Это не одно и то же, что использование процедур *Write* или *WriteLn*. Дело в том, что специально для графического режима разработаны процедуры, обеспечивающие вывод сообщений различными шрифтами в горизонтальном или вертикальном направлении, с изменением размеров и т.д. Однако в стандартных шрифтах, разработанных для этих целей фирмой *Borland*, отсутствует кириллица, что исключает вывод русскоязычных сообщений.

С другой стороны, процедуры *Write* и *WriteLn* после загрузки в память второй половины таблицы знакогенератора (а эта операция легко реализуется в адаптерах *EGA* и *VGA*) способны выводить сообщения с использованием национального алфавита, но не обладают мощными возможностями специальных процедур.

В гл.24 обсуждаются приемы, с помощью которых можно изменять стандартные шрифты, добавив в них кириллицу, и/или приспособить процедуры *Write*, *WriteLn* к выводу сообщений в графическом режиме.

Ниже описываются стандартные средства модуля *Graph* для вывода текста.

Процедура *OutText*. Выводит текстовую строку, начиная с текущего положения указателя. Заголовок:

Procedure *OutText* (Txt: **String**);

Здесь *Txt* — выводимая строка.

При горизонтальном направлении вывода указатель смещается в конец выведенного текста, при вертикальном — не меняет своего положения. Строка выводится в соответствии с установленным стилем и выравниванием. Если текст выходит за границы экрана, то при использовании штриховых шрифтов он отсекается, а в случае стандартного шрифта не выводится.

Процедура *OutTextXY*. Выводит строку, начиная с заданного места. Заголовок:

Procedure *OutTextXY* (X,Y: Integer; Txt: **String**);

Здесь *X*, *Y* — координаты точки вывода; *Txt* — выводимая строка.

Отличается от процедуры *OutText* только координатами вывода. Указатель не меняет своего положения.

Процедура *SetTextStyle*. Устанавливает стиль текстового вывода на графический экран. Заголовок:

Procedure *SetTextStyle* (Font, Direct, Size: Word);

Здесь *Font* — код (номер) шрифта; *Direct* — код направления; *Size* — код размера шрифта.

Для указания кода шрифта можно использовать следующие предварительно определенные константы:

const

DefaultFont = 0; {Точечный шрифт 8x8}
 TriplexFont = 1; {Утроенный шрифт TRIP.CHR}
 SmallFont = 2; {Уменьшенный шрифт LITT.CHR}
 SansSerifFont = 3; {Прямой шрифт SANS.CHR}
 GothicFont = 4; {Готический шрифт GOTH.CHR}

Замечу, что эти константы определяют все шрифты для версий 4.0, 5.0, 5.5 и 6.0. В версии 7.0 набор шрифтов значительно расширен, однако для новых шрифтов не предусмотрены соответствующие мнемонические константы. В этой версии помимо перечисленных Вы можете при обращении к *SetTextStyle* использовать такие номера шрифтов:

Номер	Файл	Краткое описание
-------	------	------------------

5	scri.chr	«Рукописный» шрифт
6	simp.chr	Одноштриховый шрифт типа <i>Courier</i>
7	tscr.chr	Красивый наклонный шрифт типа <i>Times Italic</i>
8	lcom.chr	Шрифт типа <i>Times Roman</i>
9	euro.chr	Шрифт типа <i>Courier</i> увеличенного размера
10	bold.chr	Крупный двухштриховый шрифт

Шрифт *DefaultFont* входит в модуль *Graph* и доступен в любой момент. Это — единственный матричный шрифт, т.е. его символы создаются из матриц 8x8 пиксел. Все остальные шрифты — векторные: их элементы формируются как совокупность векторов (штрихов), характеризующихся направлением и размером. Векторные шрифты отличаются более богатыми изобразительными возможностями, но главная их особенность заключается в легкости изменения размеров без существенного ухудшения качества изображения. Каждый из этих шрифтов размещается в отдельном дисковом файле. Если Вы собираетесь использовать какой-либо векторный шрифт, соответствующий файл должен находиться в Вашем каталоге, в противном случае вызов этого шрифта игнорируется и подключается стандартный шрифт (проблема включения шрифта в готовую программу обсуждается в п.22.9).

Замечу, что шрифт *DefaultFont* создается графическим драйвером в момент инициации графики на основании анализа текстового шрифта. Поэтому, если Ваш ПК способен выводить кириллицу в текстовом режиме, Вы сможете с помощью этого шрифта выводить русскоязычные сообщения и в графическом режиме. В остальных шрифтах эта возможность появляется только после их модификации (см. п.24.2).

Для задания направления выдачи текста можно использовать константы:

const

HorizDir = 0; {Слева направо}

VertDir = 1; {Снизу вверх}

Как видим, стандартные процедуры *OutText* и *OutTextXY* способны выводить сообщения лишь в двух возможных направлениях — слева направо или снизу вверх. Зная структуру векторных шрифтов, нетрудно построить собственные процедуры вывода, способные выводить сообщения в любом направлении (см. в п.24.3.4 процедуру *OutString* модуля *F_GrText*).

Каждый шрифт способен десятикратно изменять свои размеры. Размер выводимых символов кодируется параметром *Size*, который может иметь значение в диапазоне от 1 до 10 (точечный шрифт — в диапазоне от 1 до 32). Если значение параметра равно 0, устанавливается размер 1, если больше 10 — размер 10. Минимальный размер шрифта, при котором еще отчетливо различаются все его детали, равен 4 (для точечного шрифта — 1).

Следующая программа демонстрирует различные шрифты (рис.14.11). Их размер выбран так, чтобы строки имели приблизительно одинаковую высоту. Перед исполнением программы скопируйте все шрифтовые файлы с расширением *.CHR* в текущий каталог.

N	Name	Size	Symbols
1	TRIP	5	abcdefghijklmnopqrs
2	LITT	10	abcdefghijklmnopqr
3	SANS	4	abcdefghijklmnopqrs
4	GOTH	4	abcdefghijklmnopqrs
5	SCRI	4	abcdefghijklmnopqrs
6	SIMP	4	abcdefghijklmnopqrs
7	TSCR	4	abcdefghijklmnopqr
8	LCOM	4	abcdefghijklmnopqr
9	EURO	3	abcdefghijklmnopqr
10	BOLD	2	abcdefghijklmnopqr

Рис.14.11. Различные шрифты

Uses Graph, CRT;

const

```
FontNames: array [1..10] of String [4] =
  ('TRIP', 'LITT', 'SANS', 'GOTH',
   'SCRI', 'SIMP', 'TSCR', 'LCOM', 'EURO', 'BOLD');
```

```
Tab1 = 50;
```

```
Tab2 = 150;
```

```
Tab3 = 220;
```

var

```
d, r, Err, {Переменные для инициации графики}
```

```
Y, dY, {Ордината вывода и ее приращение}
```

```
Size, {Размер символов}
```

```
MaxFont, {Максимальный номер шрифта}
```

```
k: Integer; {Номер шрифта}
```

```
NT, SizeT, SymbT: String; {Строки вывода}
```

```
c: Char;
```

```
{-----}
```

Procedure OutTextWithTab (S1, S2, S3, S4: String);

{Выводит строки S1..S4 с учетом позиций табуляции Tab1..Tab3}

begin

```

MoveTo((Tab1-TextWidth(S1)) div 2,Y);
OutText(S1);
MoveTo(Tab1+(Tab2-Tab1-TextWidth(S2)) div 2,Y);
OutText(S2);
MoveTo(Tab2+(Tab3-Tab2-TextWidth(S3)) div 2,Y);
OutText(S3);
if S4='Symbols' then {Заголовок колонки Symbols}
MoveTo((Tab3+GetMaxX-TextWidth(S4)) div 2,Y)
else {Остальные строки}
MoveTo(Tab3+3,Y);
OutText(S4)
end;
{-----}
begin
{Иницилируем графику}
InitGraph(d,r,'');
Err := GraphResult;
if Err<>grOk then
WriteLn(GraphErrorMsg(Err))
else
begin
{Определяем количество шрифтов:}
{$IFDEF VER70}
MaxFont := 10;
{$ELSE}
MaxFont := 4;
{$ENDIF}
SetTextStyle(1,0,4);
Y := 0;
OutTextWithTab('N','Name','Size','Symbols');
{Определяем высоту Y линии заголовка}
Y := 4*TextHeight('Z') div 3;
Line(0,Y,GetMaxX,Y);
{Определяем начало Y таблицы и высоту dY каждой строки}
Y := 3*TextHeight('Z') div 2;
dY := (GetMaxY-Y) div (MaxFont);
{Готовим строку символов}
SymbT := '';
for c := 'a' to 'z' do
SymbT := SymbT+c;
{Цикл вывода строк таблицы}
for k := 1 to MaxFont do
begin
Size := 0;
{Увеличиваем размер до тех пор, пока высота строки
не станет приблизительно равна dY}
repeat

```

```

        inc(Size);
        SetTextStyle(k,0,Size+1);
    until (TextHeight('Z')>=dY) or (Size=10)
        or (TextWidth(FontNames[k])>(Tab2-Tab1));
    {Готовим номер NT и размер SizeT шрифта}
    Str(k,NT);
    Str(Size,SizeT);
    {Выводим строку таблицы}
    SetTextStyle(k,HorizDir,Size);
    OutTextWithTab(NT,FontNames[k],SizeT,SymbT);
    inc(Y,dY)
end;
{Рисуем линии рамки}
Rectangle(0,0,GetMaxX,GetMaxY);
Line(Tab1,0,Tab1,GetMaxY);
Line(Tab2,0,Tab2,GetMaxY);
Line(Tab3,0,Tab3,GetMaxY);
{Ждем инициативы пользователя}
ReadLn;
CloseGraph
end
end.

```

Процедура SetTextJustify. Задаёт выравнивание выводимого текста по отношению к текущему положению указателя или к заданным координатам. Заголовок:

```
Procedure SetTextJustify(Horiz,Vert: Word);
```

Здесь *Horiz* – горизонтальное выравнивание; *Vert* – вертикальное выравнивание.

Выравнивание определяет как будет размещаться текст – левее или правее указанного места, выше, ниже или по центру. Здесь можно использовать такие константы:

```
const
```

```

LeftText  = 0; {Указатель слева от текста}
CenterText= 1; {Симметрично слева и справа,верху и снизу}
RightText = 2; {Указатель справа от текста}
BottomText= 0; {Указатель снизу от текста}
TopText   = 2; {Указатель сверху от текста}

```

Обратите внимание на неудачные, с моей точки зрения, имена мнемонических констант: если, например, Вы зададите *LeftText*, что в переводе означает «ЛевыйТекст», сообщение будет расположено справа от текущего положения указателя (при выводе процедурой *OutTextXY* – справа от заданных координат). Также «наоборот» трактуются и остальные константы.

Следующая программа иллюстрирует различные способы выравнивания относительно центра графического экрана (рис.14.12).

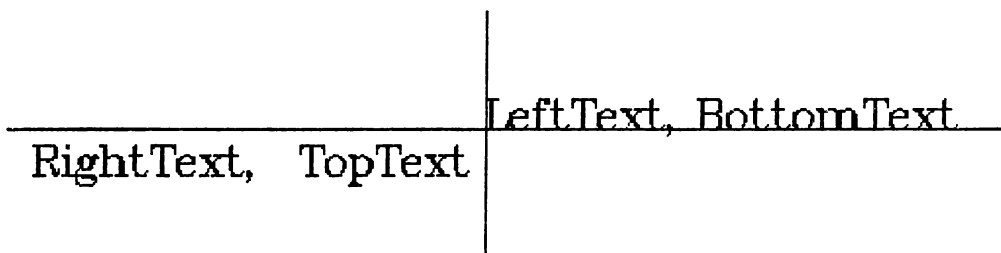


Рис.14.12. Вывод текста

```

Uses Graph, CRT;
var
  d, r, e : Integer;
begin
  {Иницилируем графику}
  d := Detect;
  InitGraph(d, r, '');
  e := GraphResult;
  if e <> grOk then
    WriteLn(GraphErrorMsg(e))
  else
    begin
      {Выводим перекрестие линий в центре экрана}
      Line(0,GetMaxY div 2,GetMaxX,GetMaxY div 2);
      Line(GetMaxX div 2,0,GetMaxX div 2,GetMaxY);
      {Располагаем текст справа и сверху от центра}
      SetTextStyle(TriplexFont,HorizDir,3);
      SetTextJustify(LeftText,BottomText);
      OutTextXY (GetMaxX div 2, GetMaxY div 2,
        'LeftText,BottomText');
      {Располагаем текст слева и снизу}
      SetTextJustify (RightText, TopText);
      OutTextXY (GetMaxX div 2, GetMaxY div 2,
        'RightText, TopText');
      if ReadKey=#0 then d := ord(ReadKey);
      CloseGraph
    end
  end.

```

Процедура SetUserCharSize. Изменяет размер выводимых символов в соответствии с заданными пропорциями. Заголовок:

```
Procedure SetUserCharSize(X1,X2,Y1,Y2: Word);
```

Здесь X1...Y2 — выражения типа *Word*, определяющие пропорции по горизонтали и вертикали.

Процедура применяется только по отношению к векторным шрифтам. Пропорции задают масштабный коэффициент, показывающий во сколько

раз увеличится ширина и высота выводимых символов по отношению к стандартно заданным значениям. Коэффициент по горизонтали находится как отношение $X1$ к $X2$, по вертикали — как отношение $Y1$ к $Y2$. Чтобы, например, удвоить ширину символов, необходимо задать $X1=2$ и $X2=1$. Стандартный размер символов устанавливается процедурой *SetTextStyle*, которая отменяет предшествующее ей обращение к *SetUserCharSize*.

В следующем примере демонстрируется изменение пропорций уменьшенного шрифта.

```
Uses Graph, CRT;
var
  d, r, e : Integer;
begin
  {Инициализируем графику}
  d := Detect; InitGraph(d, r, '');
  e := GraphResult;
  if e <> grOk then
    WriteLn(GraphErrorMsg(e))
  else
    begin
      MoveTo (0, GetMaxY div 2);
      SetTextStyle (SmallFont, HorizDir, 5);
      SetTextJustify (LeftText, BottomText);
      {Выводим сообщение стандартной высотой 5}
      OutText ('Normal Width,');
      {Удваиваем ширину шрифта}
      SetUserCharSize (2, 1, 1, 1);
      OutText (' Double Width, ');
      {Удваиваем высоту, возвращаем стандартную ширину}
      SetUserCharSize (1, 1, 2, 1);
      OutText ('Double Height,');
      SetUserCharSize (2, 1, 2, 1);
      OutText (' Double Width and Height');
      if ReadKey=#0 then d := ord(ReadKey);
      CloseGraph
    end
  end.
end.
```

Функция TextWidth. Возвращает длину в пикселах выводимой текстовой строки. Заголовок:

```
Function TextWidth(Txt: String): Word;
```

Учитываются текущий стиль вывода и коэффициенты изменения размеров символов, заданные соответственно процедурами *SetTextStyle* и *SetUserCharSize*.

Функция TextHeight. Возвращает высоту шрифта в пикселах. Заголовок:

```
Function TextHeight(Txt: String): Word;
```

Процедура GetTextSettings. Возвращает текущий стиль и выравнивание текста. Заголовок:

```
Procedure GetTextSettins(var TextInfo: TextSettingsType);
```

Здесь *TextInfo* — переменная типа *TextSettingsType*, который в модуле *Graph* определен следующим образом:

```
type
  TextSettingsType = record
    Font      : Word; {Номер шрифта}
    Direction: Word; {Направление}
    CharSize  : Word; {Код размера}
    Horiz     : Word; {Горизонтальное выравнивание}
    Vert      : Word  {Вертикальное выравнивание}
  end;
```

Функция InstallUserFont. Позволяет программе использовать нестандартный векторный шрифт. Заголовок функции:

```
Function InstallUserFont(FileName: String): Integer;
```

Здесь *FileName* — имя файла, содержащего векторный шрифт.

Как уже говорилось, в стандартную поставку Турбо Паскаля версий 4.0 — 6.0 включены три векторных шрифта, для версии 7.0 — 10. Функция *InstallUserFont* позволяет расширить этот набор. Файл *FileName* должен строиться по специальным правилам, подробно обсуждаемым в п.23.2. Функция возвращает идентификационный номер нестандартного шрифта, который может использоваться при обращении к процедуре *SetTextStyle*.

Функция InstallUserDriver. Включает нестандартный графический драйвер в систему *BGI*—драйверов. Заголовок функции:

```
Function InstallUserDriver(FileName: String;  
  AutoDetectPtr: Pointer): Integer;
```

Здесь *FileName* — имя файла, содержащего программу драйвера; *AutoDetectPtr* — адрес точки входа в специальную процедуру автоопределения типа дисплея, которая в числе прочих процедур должна входить в состав драйвера.

Эта функция расширяет и без того достаточно обширный набор стандартных графических драйверов и предназначена в основном для разработчиков аппаратных средств.

14.9. ВКЛЮЧЕНИЕ ДРАЙВЕРА И ШРИФТОВ В ТЕЛО ПРОГРАММЫ

В Турбо Паскале имеется возможность включения графического драйвера и штриховых шрифтов непосредственно в тело программы. Такое включение делает программу независимой от местоположения и наличия

на диске драйверов и шрифтов, а также ускоряет подготовку графических программ к работе (шрифты и драйвер загружаются вместе с программой).

Включение драйвера и шрифтов осуществляется по следующей общей схеме. Сначала с помощью вспомогательной программы *BINOBJ.EXE*, входящей в комплект поставки Турбо Паскаля, драйвер и шрифты преобразуются в *OBJ*-файл (файл с расширением *.OBJ*). Для этого вне среды Турбо Паскаля необходимо вызвать утилиту *BINOBJ* с тремя параметрами: именем преобразуемого файла, именем получаемого *OBJ*-файла и глобальным именем процедуры. Эти имена, в принципе, могут быть произвольными, правильными для *MS-DOS* именами. Например:

```
c:\tp\binobj cga.bgi cga cgadrv
```

В результате такого обращения из каталога *TP* на диске *C* будет вызвана программа *BINOBJ* и ей будут переданы следующие параметры:

CGA.BGI — имя файла с преобразуемым драйвером;

CGA — имя файла с расширением *.OBJ*, т.е. *CGA.OBJ*, который будет получен в результате исполнения программы *BINOBJ*;

CGADRV — глобальное имя, под которым этот драйвер будет известен программе.

После этого можно написать следующий фрагмент программы:

```
Uses Graph;
Procedure CGADRV; external;
{$L CGA.OBJ}
var
  d, r, e : Integer;
begin
  if RegisterBGIDriver (@CGADRV) < 0 then
    begin
      WriteLn ('Ошибка при регистрации драйвера');
      halt
    end;
  d := CGA; r := CGAHi;
  InitGraph (d, r, '');
  .....
```

Как видно из этого примера, в программе объявляется внешняя процедура с именем *CGADRV* (глобальное имя, указанное при обращении к *BINOBJ*), причем дается директива компилятору отыскать в текущем каталоге и загрузить файл *CGA.OBJ*, в котором находится эта процедура. Затем осуществляется регистрация драйвера путем обращения к функции *RegisterBGIDriver*. Единственным параметром этой функции является адрес начала драйвера в памяти (*@CGADRV*). Функция возвращает значение типа *Integer*, которое служит для контроля правильности завершения процедуры регистрации драйвера: если это значение меньше нуля, обнаружена ошибка, в противном случае функция возвращает номер зарегистрированного драйвера. В примере контролируется правильность регистрации драйвера и, если ошибка не обнаружена иницируется графический режим работы экрана.

Аналогичным образом можно присоединить к программе стандартные штриховые шрифты (матричный шрифт 8x8 входит в состав модуля *Graph* и поэтому присоединять его не надо). Присоединение шрифта строится по описанной схеме за тем исключением, что для его регистрации вызывается функция *RegisterBGIFont*. Например, после преобразования

```
c:\Pascal\binobj litt.chr litt litt
```

можно использовать операторы

```
.....
Procedure Litt; External;
{$L Litt.obj}
.....
if RegisterBGIFont (@litt) < 0 then ...
```

Обратите внимание: регистрация и драйвера, и шрифтов должна предшествовать инициации графического режима.

Регистрировать можно также драйверы (шрифты), которые не компилируются вместе с программой, а загружаются в динамическую память. Например:

```
Uses Graph;
var
  p: Pointer;
  f: file;
begin
  Assign(f, 'Litt.chr');           {Открываем файл}
  Reset(f, 1);                    {LITT.CHR для чтения}
  GetMem(p, FileSize(f));         {Резервируем для него
                                  область кучи нужного размера}
  BlockRead(f, p^, FileSize(f));  {Читаем файл}
  WriteLn(RegisterBGIFont(p))      {Регистрируем шрифт}
end.
```

ЧАСТЬ 2

**БИБЛИОТЕКА
TURBO VISION**

ВВЕДЕНИЕ В TURBO VISION

В этой главе мы попробуем разработать программу, которая использует некоторые возможности Turbo Vision. Пусть, например, нам необходимо создать простейшую информационную систему — нечто вроде электронной записной книжки. Предполагается, что данные, используемые этой системой, будут храниться в виде записей в дисковом файле. Наша задача — разработать удобную диалоговую программу, облегчающую доступ к файловым данным.

Разработка программы, разумеется, не является самоцелью — ведь для нас это только повод для конкретного знакомства с Turbo Vision. Поэтому мы будем создавать программу постепенно, каждый раз фиксируя достигнутые результаты. Если Вас интересует собственно информационная программа, используйте ее окончательный вариант, приведенный в прил.П.5.4

15.1. ПРОСТЕЙШАЯ ПРОГРАММА В TURBO VISION

Работа большинства прикладных программ проходит в три этапа: подготовка к работе, собственно работа и, наконец, ее завершение. В нашем случае к подготовительному этапу можно отнести такие действия, как анализ существования файла данных и его (файла) подготовка к работе. На этапе завершения мы должны обеспечить необходимые действия по сохранению файла. Все остальные действия относятся к среднему этапу. С учетом этого можно написать следующую простейшую программу:

```
begin
    {Подготовить работу программы}
    {Выполнить необходимые действия}
    {Завершить исполнение программы}
end.
```

Если Вы попытаетесь выполнить эту программу, ничего не произойдет — ведь мы еще никак не конкретизировали необходимые действия. Так обстоит дело в Турбо Паскале, но не так — в Turbo Vision! Для любой прикладной программы Turbo Vision сразу же создает некоторую минимальную программную реализацию, которую Вы можете затем постепенно наращивать в ходе детализации программы. Вот начальный вариант программы с использованием Turbo Vision:

```
Uses App; {Используется модуль APP библиотеки Turbo Vision}
var
    Notebook: TApplication;
begin
    Notebook.Init; {Подготовить работу программы}
    Notebook.Run;  {Выполнить необходимые действия}
```

```
Notebook.Done {Завершить исполнение программы}  
end.
```

В этой программе объявляется использование стандартного для Turbo Vision модуля *App* (от *application* — приложение, прикладная программа). Такое объявление открывает доступ прикладной программе к мощным возможностям Turbo Vision. Чтобы использовать эти возможности, мы объявили переменную *Notebook* (*notebook* — записная книжка) типа *TApplication*. Как Вы вскоре заметите, на букву *T* в Turbo Vision начинаются идентификаторы объектов. Таким образом, *Notebook* — это экземпляр объекта *TApplication*, т.е. объединение данных (полей) и методов обработки этих данных (процедур, функций, конструкторов, деструкторов). В объекте *TApplication* предусмотрены методы *Init*, *Run* и *Done*. Вызов этих методов и составляет исполняемую часть нашей программы. Если Вы подготовите и запустите программу, на экране ПК появится изображение, показанное на рис.15.1.

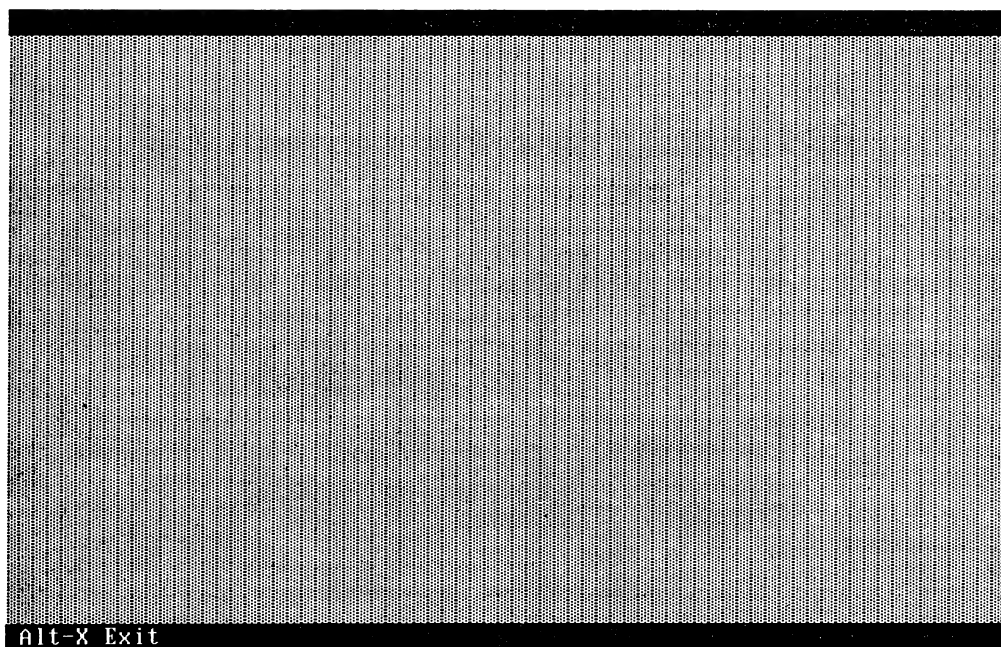


Рис.15.1. Вид экрана для простейшей программы

Для выхода из программы необходимо, как это следует из надписи в левом нижнем углу экрана, нажать *Alt-X* или подвести к этой надписи указатель мыши (если, разумеется, Ваш ПК оснащен этим устройством) и нажать ее левую кнопку.

Как видите, даже простейшая программа «знает», как создать экран, распознает команду *Alt-X* и может работать с мышью. Совсем не плохо для трех исполняемых операторов, не так ли? Такие возможности дос — тупны потому, что в объекте *TApplication* предусмотрены соответствующие методы. В этом смысле использование объектов напоминает использование

подпрограмм из библиотек. Однако в отличие от подпрограммы любой объект имеет все необходимые ему данные. Говоря об объектах, я часто буду использовать такие слова, как «знает», «умеет», «может», подчеркивая тем самым главную отличительную особенность объектов от традиционных подпрограмм — их «разумность»: последовательное проведение в жизнь принципа инкапсуляции (объединения) данных и всех необходимых для их обработки методов придает объекту определенную независимость от других элементов программы; объекты как бы «живут» в программе своей независимой жизнью.

Простейшая программа не может выполнять никаких других действий, кроме уже перечисленных, так как именно эти действия запрограммированы в методах *Init* и *Run* объекта *TApplication*. В ходе их выполнения на экране создается изображение, имеющее три зоны: верхняя строка, нижняя строка и вся остальная часть экрана. Верхняя строка обычно используется для размещения опций главного меню (не забывайте, что Turbo Vision — это оболочка для диалоговых программ!). Нижняя строка — строка статуса: в ней указываются так называемые командные клавиши, т.е. клавиши или комбинации клавиш, которые вызывают нужные действия без перехода к промежуточному диалогу. Вся остальная часть экрана составляет «рабочий стол» программы — сюда будут помещаться сообщения, здесь будут размещаться окна, «выпадающие» меню (меню нижнего уровня) и т.п.

15.2. ФОРМИРОВАНИЕ СТРОКИ СТАТУСА

Стандартный вид экрана, показанный на рис.15.1, можно изменять. Попробуем придать ему некоторый специфический для нашей программы вид. Например, заменим в строке статуса стандартное сообщение

Alt-X Exit

на русифицированное

Alt-X Выход

Таким образом, нам необходимо *модифицировать* стандартное поведение объекта *Notebook*. Для этого мы должны отыскать в типе *TApplication* метод, ответственный за создание строки статуса. Если мы обратимся к прил.П6, то обнаружим, что объект типа *TApplication* содержит методы *Init* и *Done*, с помощью которых создаются и уничтожаются экземпляры объекта, но в нем нет метода, ответственного за строку статуса. Однако из таблицы наследования нетрудно определить, что этот метод (*InitStatusLine*) он наследует от своего родителя *TProgram*. Как изменить работу метода? В рамках объектно-ориентированной библиотеки для этого поступают следующим образом: объявляется объект-потомок от стандартного объекта, поведение которого необходимо изменить, и в новом объекте описывается свой метод, ответственный за это поведение.

Изменим программу следующим образом:

```
Uses App, Objects, Menus, Drivers, Views;  
type
```

```

TNotebook = object (TApplication) {Создаем объект-потомок
                                     от TApplication;
  Procedure InitStatusLine; Virtual; {Перекрываем старый
                                     метод InitStatusLine новым}
end;
{-----}
Procedure TNotebook.InitStatusLine;
{Описание нового метода, с помощью которого
создается строка статуса}
var
  R: TRect; {Границы строки статуса}
begin
  GetExtent(R);           {Получаем в R координаты всего экрана}
  R.A.Y := pred(R.B.Y);   {Помещаем в R координаты строки
                           статуса}

  {Создаем строку статуса;}
  StatusLine := New(PStatusLine, Init(R,
  {Определяем один вариант строки статуса;}
  NewStatusDef(0, $FFFF, {Устанавливаем для этого варианта
                           максимальный диапазон контекстной
                           справочной службы}
  {Определяем единственную клавишу Alt-X;}
  NewStatusKey('~Alt-X~ Выход', kbAltX, cmQuit,
  NIL), {Нет других клавиш}
  NIL) {Нет других строк статуса}
  ));
end; {TNotebook.InitStatusLine}
{-----}
var
  Notebook: TNotebook; {Изменен тип переменной!}
begin
  Notebook.Init;
  Notebook.Run;
  Notebook.Done
end.

```

Как видим, программа сразу же усложнилась. Во-первых, в ней используются идентификаторы, которые определены в других модулях Turbo Vision, — эти модули мы перечислили в предложении *Uses*. Во-вторых, нам потребовалось объявить новый объект *TNotebook* как потомок от объекта *TApplication*. Объект-потомок наследует от своего объекта-родителя все поля и методы и при необходимости может их дополнять своими полями и методами, а также перекрывать методы родителя. Как раз для того, чтобы перекрыть унаследованный от *TProgram* стандартный метод *InitStatusLine*, ответственный за создание строки статуса, нам и понадобилось объявление нового типа *TNotebook*. Строка

```
Procedure InitStatusLine; Virtual;
```

в объявлении этого типа указывает, что новый объект будет пользоваться одноименным, но иным, чем объект-родитель, методом. Возможность замены методов на одноименные, но с другим содержанием называется полиморфизмом.

Процедура *TNotebook.InitStatusLine* раскрывает суть нового метода. В ней используется обращение к методам *NewStatusDef* и *NewStatusKey*, с помощью которых создается динамический объект типа *TStatusLine*. Программа *TApplication* обращается к методам этого объекта для обслуживания строки статуса.

Turbo Vision позволяет определять несколько вариантов строки статуса. Каждый вариант создается с помощью метода *NewStatusDef*. В зависимости от текущего состояния программы (от *контекста* программы) Turbo Vision автоматически помещает в строку статуса нужный вариант. Так как в нашей программе используется единственный вариант строки статуса, мы указали максимально возможный диапазон контекста программы при обращении к методу *NewStatusDef*.

С помощью метода *NewStatusKey* в строке статуса определяется очередная командная клавиша. При обращении к методу сначала указывается текст, высвечиваемый в строке статуса, причем символом «`~`» выделяется та часть сообщения, которая будет подсвечена в строке статуса другим цветом: таким способом в Turbo Vision указываются командные клавиши. Идентификатор *kbAltX* задает комбинацию клавиш, а *cmQuit* — связанную с ней команду.

В программе объявляется переменная *R* типа *TRect*. С помощью такого типа переменных программист задает координаты прямоугольного участка экрана. Эта переменная необходима нам для указания того места на экране, где будет помещено сообщение

Alt-X Выход

определяющее командные клавиши *Alt-X*. Для правильного задания этих координат мы использовали два предложения:

```
GetExtent(R);  
R.A.Y := pred(R.B.Y);
```

В первом вызывается стандартный метод Turbo Vision *GetExtent*, с помощью которого в *R* помещаются координаты доступной в данный момент части экрана. Во втором — номер той строки (строки статуса), куда будет выводиться сообщение. В Turbo Vision тип *TRect* объявляется в виде следующей записи:

```
type  
TRect = record  
  A: record {Координаты верхнего левого угла}  
    X: Byte;  
    Y: Byte  
  end;  
  B: record {Координаты правого нижнего угла}  
    X: Byte;
```

```

Y: Byte
end
end;

```

Таким образом, второе предложение лишь уменьшает на единицу вертикальную координату самой нижней доступной строки и устанавливает полученное значение в поле *R.A.Y* (это поле задает вертикальную координату верхнего левого угла прямоугольного участка). Заметим, что в Turbo Vision минимальные координаты задаются значением 0, в то время как в стандартном модуле *CRT* Турбо Паскаля минимальные координаты имеют значение 1.

Обратите внимание на характерный прием, широко используемый в Turbo Vision: при обращении к методам *NewStatusDef* и *NewStatusKey* последним параметром указывается переменная типа *Pointer*. Внутри методов эта переменная трактуется как ссылка на новый метод, что позволяет организовать цепочку последовательных определений. Вложенная последовательность вызовов заканчивается зарезервированной константой *NIL*, указывающей на конец цепочки. Если бы мы, например, захотели добавить в строку статуса определение клавиши *F10*, связав ее с закрытием активного окна, мы могли бы использовать такую конструкцию:

```

NewStatusDef(0, $FFFF,
  NewStatusKey('~Alt-X~ Выход', kbAltX, cmQuit,
    NewStatusKey('~F10~ Закрыть окно', kbF10, cmClose,
      NIL)),      {Нет других клавиш}
  NIL)           {Нет других определений}

```

Описанный пример позволяет нам сделать очень важный вывод:

Чтобы модифицировать стандартное поведение объекта, необходимо: создать объект-потомок от этого объекта и перекрыть в нем нужный метод.

15.3. ФОРМИРОВАНИЕ МЕНЮ

Вернемся к смысловой части нашего примера и подумаем о том, какие еще свойства следует придать программе. Поскольку мы предполагаем работу с файлом, можно включить в программу код, реализующий строку меню с опцией «Файл», связав с этой опцией такие действия, как открытие уже существующего файла данных и/или создание нового. Здесь же можно предусмотреть возможность альтернативного выхода из программы. Кроме того, в главное меню следует поместить еще одну опцию, назовем ее «Работа». Эта опция должна открыть доступ к содержательной части программы. С учетом сказанного программу нужно дополнить следующими строками:

```

const
  {Команды для обработчиков событий:}
  cmWork      = 203; {Обработать данные}

```

```

cmDOS      = 204; {Временно выйти в ДОС}
WinCom1: TCommandSet = [cmSave,cmWork]; {Множество
                                         временно недоступных команд}

```

Эти строки следует вставить сразу после предложения *Uses*; они определяют коды команд, которые будут затем использоваться для вызова соответствующих частей программы. Кроме того, объявление объекта *TNotebook* нужно дополнить строкой

```

type
  TNotebook = object (TApplication)
    ....
    Procedure InitMenuBar; Virtual; {Перекрываем стандартный
                                     метод InitMenuBar}
  end;

```

в которой перекрывается прежний метод *InitMenuBar*, ответственный за формирование строки меню. И, наконец, в разделе объявлений программы следует поместить описание метода *InitMenuBar* и видоизменить описание метода *InitStatusLine*:

```

Procedure TNotebook.InitMenuBar;
{Создание верхнего меню}
var
  R: TRect;
begin
  GetExtent(R);
  R.B.Y := succ(R.A.Y); {R - координаты строки меню}
  MenuBar := New(PMenuBar, Init(R,
    NewMenu(
      {Создаем меню}
      {Первый элемент нового меню представляет собой
      подменю (меню второго уровня). Создаем его}
      NewSubMenu('~F~/ Файл', hcNoContext,
        {Описываем элемент главного меню}
        NewMenu(
          {Создаем подменю}
          NewItem(
            {Первый элемент}
            '~1~/ Открыть', 'F3', kbF3, cmOpen, hcNoContext,
            NewItem(
              {Второй элемент}
              '~2~/ Закрыть', 'F2', kbF2, cmSave, hcNoContext,
              NewItem(
                {Третий элемент}
                '~3~/ Сменить диск', '', 0, cmChangeDir, hcNoContext,
              NewLine(
                {Строка-разделитель}
                NewItem('~4~/ Вызов ДОС', '', 0, cmDOSShell,
                  hcNoContext,
                  NewItem('~5~/ Конец работы', 'Alt-X',
                    kbAltX, cmQuit, hcNoContext,
                    NIL)))))) {Нет других элементов подменю}
        ),
        {Создаем второй элемент главного меню}
        NewItem('~W~/ Работа', '', kbF4, cmWork, hcNoContext,

```

```

    NIL) {Нет других элементов главного меню}
  ))))
end; {TNotebook.InitMenuBar}
{-----}
Procedure TNotebook.InitStatusLine;
{Формирует строку статуса}
var
  R: TRect;           {Границы строки статуса}
begin
  GetExtent(R);      {Получаем в R координаты всего экрана}
  R.A.Y := pred(R.B.Y);
  StatusLine := New(PStatusLine,
    Init(R,           {Создаем строку статуса}
      NewStatusDef(0, $FFFF, {Устанавливаем максимальный
        диапазон контекстной справочной службы}
      NewStatusKey('~Alt-X~ Выход', kbAltX, cmQuit,
      NewStatusKey('~F2~ Закрыть', kbF2, cmSave,
      NewStatusKey('~F3~ Открыть', kbF3, cmOpen,
      NewStatusKey('~F4~ Работа', kbF4, cmWork,
      NewStatusKey('~F10~ Меню', kbF10, cmMenu,
      NIL))))) ,      {Нет других клавиш}
    NIL)              {Нет других определений}
  ));
  DisableCommands(WinCom1) {Запрещаем недоступные команды}
end; {TNotebook.InitStatusLine}

```

В новом варианте программы мы продвинулись дальше по пути конкретизации ее действий. Если Вы запустите программу и нажмете клавиши *Alt-F* (вызов опции «Файл» главного меню), на экране появится изображение, показанное на рис.15.2.

Определение опций меню во многом напоминает определение командных клавиш в строке статуса. Отличие заключается лишь в том, что с любой опцией меню может быть при необходимости связана встроенная справочная служба. В нашей программе мы не используем эту возможность, для чего задаем стандартный идентификатор *hcNoContext* (нет контекстно-зависимой справки) при описании каждой опции.

Подобно клавишам строки статуса командные клавиши меню выделяются символом «~». Заметим, что не имеет смысла назначать в качестве командных клавиш клавиши кириллицы, так как при их анализе Turbo Vision игнорирует коды 128...255. Если бы, например, мы задали в качестве командной клавиши для опции «Файл» клавишу «Ф», нажатие *Alt-Ф* не вызвало бы развертывания подменю, связанного с этой опцией (как и в Турбо Паскале, в Turbo Vision опции главного меню вызываются комбинацией *Alt-<клавиша>*, а опции меню нижнего уровня — просто нажатием нужной командной клавиши).

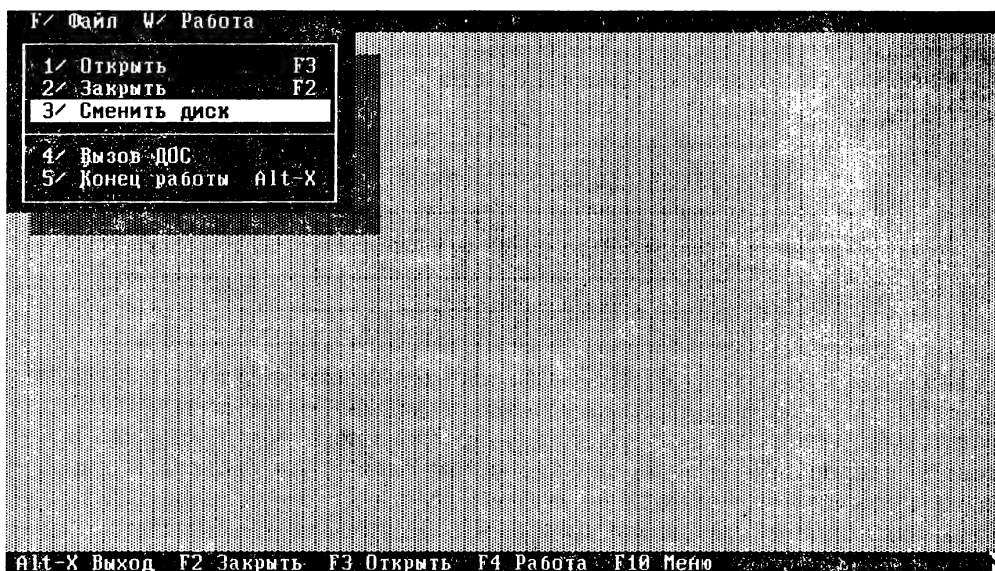


Рис.15.2. Вид окна с развернутым меню опции Файл

15.4. КОМАНДЫ

Необходимо пояснить назначение вновь введенных констант *cmXXXX*. Это так называемые команды, точнее их коды (шифр). Сразу же замечу, что префикс *cm* в идентификаторах команд не является следствием каких-либо требований со стороны Turbo Vision, просто он принят для предопределенных (стандартных) команд, таких как *cmQuit* и *cmClose*. Вновь вводимые команды не являются предопределенными, при их описании я заимствовал стандартный префикс только по соображениям стилистики.

Что же такое команды Turbo Vision? Внимательный анализ предыдущего варианта программы показывает, что эти коды еще никак не используются, они понадобились лишь для синтаксически правильного обращения к стандартным методам инициации строк меню и статуса. В новом варианте программной реализации Вы можете вызвать любую опцию главного меню или нажать любую командную клавишу — это не приведет ни к каким последствиям: пока работает только команда *Alt-X*, завершающая работу программы, и клавиши *F2*, *F3* и *F10*. Происходит это потому, что эти клавиши мы связали со стандартными командами *cmQuit*, *cmSave*, *cmOpen* и *cmMenu* и обрабатываются они где-то внутри Turbo Vision. Новые команды не известны системе, и их обработку мы должны взять на себя.

Как мы увидим дальше, в Turbo Vision есть средства контроля командных клавиш. Эти средства определяют факт нажатия на клавишу с помощью генерации кода соответствующей команды, который (код) будет в этом случае передан нашей программе. Таким образом, обработка команд заключается в расшифровке получаемых от Turbo Vision кодов и передаче управления соответствующим частям программы.

Для шифровки команд в Turbo Vision используется 16-разрядное слово, что позволяет определить до 65535 различных команд. Некоторые из этих кодов зарезервированы для использования внутри Turbo Vision, остальные доступны программисту:

Код команды	Зарезервировано	Можно запретить
0...99	Да	Да
100...255	Нет	Да
256...999	Да	Нет
1000...65535	Нет	Нет

Команды с кодами от 0 до 255 при необходимости могут быть временно запрещены, остальные команды запретить нельзя — вот почему используется два диапазона доступных для программиста кодов команд. Временное запрещение команд связано с тем очевидным свойством диалоговых программ, что отнюдь не любая команда может исполняться в каком-либо возможном состоянии программы. Например, бессмысленно использовать команду «Закрыть файл», если файл еще не открыт. Наоборот, если файл уже открыт, команда «Открыть файл» может стать временно недоступной пользователю. Механизм маскирования (временного запрещения) команд позволяет избавиться от многочисленных проверок контекстуальной корректности тех или иных команд: программист может их запретить впредь до наступления какого-либо события, а запрещенные команды игнорируются средствами Turbo Vision и в программу пользователя не передаются.

В нашей программе имеет смысл запретить команды *cmSave* и *cmWork* до тех пор, пока пользователь не откроет нужный файл с данными. Запрет команд достигается обращением к стандартной процедуре *DisableCommands* (см. предыдущий вариант программы). Указанные в обращении к ней команды задаются в виде множества кодов (мощность любого множества в Турбо Паскале не может превышать 256, вот почему могут быть запрещены только первые 256 команд) и становятся недоступными впредь до обращения к процедуре *EnableCommands* (разрешить команды).

Запрещенные опции меню (как и временно недоступные командные клавиши) выделяются на экране оттенком (пониженной яркостью).

15.5. СОБЫТИЯ И ИХ ОБРАБОТКА

Весьма важным принципом Turbo Vision является принцип отделения процесса создания видимых изображений от процесса обработки данных. Это означает, что все действия по созданию разнообразных окон, меню и прочих видимых элементов можно осуществлять, не заботясь о тех командах (действиях пользователя), которые будут связаны с ними. Именно так мы поступили при определении меню и строки статуса — коды команд дают возможность распознать соответствующие действия пользователя, однако сами эти действия пока еще никак не раскрыты. И наоборот, мы можем разрабатывать части программы, ответственные за обработку

действий пользователя, не связывая прямо эти части с созданием нужных видимых элементов.

Turbo Vision поддерживает два возможных способа действия пользователя — с помощью клавиш клавиатуры и с помощью мыши. Любое такое действие пользователя с точки зрения Turbo Vision приводит к появлению *события*, т.е. к созданию небольшого информационного пакета, описывающего вновь возникшую ситуацию. События распространяются от одной части программы к другой до тех пор, пока не обнаружится подпрограмма, ответственная за обработку данного события. Эта подпрограмма обычно очищает информационный пакет и таким образом блокирует дальнейшее перемещение события.

Пожалуй, именно механизм событий кардинально отличает Turbo Vision от других библиотек Турбо Паскаля. На первых порах это может вызвать определенные трудности, связанные с отладкой программ. Принцип независимости обработки событий от процесса создания видимых элементов приводит фактически к появлению двух параллельных процессов в рамках одной программы: процесса создания видимых элементов и процесса обработки событий. Говоря о программах Turbo Vision, следует помнить, что эти программы *управляются событиями*. Их трассировка (прослеживание работы) в среде Турбо Паскаль обычно достигается установкой и использованием контрольных точек.

Подпрограммы, ответственные за обработку действий пользователя, называются *обработчиками событий*. Любой стандартный для Turbo Vision объект, обеспечивающий создание видимого элемента, имеет собственный обработчик событий (виртуальный метод *HandleEvent*), который Вы можете перекрыть своим собственным методом, если Вас не устраивает стандартная реакция объекта на то или иное событие. Существует такой метод и в объекте *TNotebook*. По умолчанию этот объект использует обработчик событий, унаследованный им от объекта-родителя *TApplication*. Стандартный обработчик знает, как реагировать на команды *cmQuit* и *cmMenu*, но ему не известны новые команды *cmWork*, *cmOpenFile* и другие. Чтобы программа смогла правильно обработать эти команды, мы должны перекрыть стандартный метод *HandleEvent* объекта *TNotebook* новым. Добавим в описание объекта *TNotebook* еще одну строку

type

```
TNotebook = object (TApplication)
```

```
.....
```

```
  Procedure HandleEvent(var Event: TEvent); Virtual;  
end;
```

и поместим в раздел объявлений текст новой подпрограммы:

```
Procedure TNotebook.HandleEvent(var Event: TEvent);
```

```
{Обработчик событий программы}
```

```
begin {TNotebook.HandleEvent}
```

```
  Inherited HandleEvent(Event);
```

```
{Обработка стандартных  
команд cmQuit и cmMenu}
```

```
  if Event.Wnat = evCommand then
```

```

case Event.Command of
  {Обработка новых команд:}
  cmOpen      : FileOpen;      {Открыть файл}
  cmSave      : FileSave;      {Закрыть файл}
  cmChangeDir : ChangeDir;     {Сменить диск}
  cmDOSShell  : DOSCall;       {Временный выход в ДОС}
  cmWork      : Work;          {Обработать данные}
else
  exit                    {Не обрабатывать другие команды}
end;
ClearEvent(Event)        {Очистить событие после обработки}
end; {TNotebook.HandleEvent}

```

Чтобы новый вариант программы можно было выполнить, следует предусмотреть «заглушки» для несуществующих пока процедур *FileOpen*, *FileSave* и т.д. Например:

```

Procedure FileOpen;
begin
end;

```

Поведение вновь созданного варианта программы внешне ничем не отличается от предыдущего: также будут созданы меню, строка статуса и основное поле экрана, программа по-прежнему будет распознавать команды *Alt-X* и *F10*. Однако теперь она будет реагировать и на новые команды. Чтобы убедиться в этом, установите контрольные точки в заглушках *FileOpen* и *FileSave* и запустите программу вновь: нажатие на клавишу *F3* вызовет останов в контрольной точке *FileOpen* — ведь именно с этой клавишей мы связали команду *cmOpen* в процедуре *InitStatusLine*, в то время как нажатие на клавишу *F2* не приведет к срабатыванию контрольной точки *FileSave*, поскольку команда *cmSave* пока еще запрещена и обработчик *HandleEvent* ее просто не «увидит».

Чтобы использовать нестандартные команды меню или строки статуса, мы должны перекрыть обработчик событий программы, в новом обработчике выделить из потока событий команды и распознать их коды.

Чтобы стали более понятны действия обработчика событий, отметим, что тип *TEvent* в Turbo Vision определен как запись такого вида:

```

type
  TEvent = record
    What: Word;                {Определяет тип события}
    case Word of
      evMouse: (                {"Пустое" событие}
        Buttons: Byte;          {Событие от мыши;}
        Double: Boolean;        {Состояние кнопок}
      )

```

```

                                нажатия кнопки мыши}
Where: TPoint);                {Координаты курсора мыши}
evKeyDown: (                   {Событие от клавиатуры:}
  case Integer of
    0: (KeyCode: Word);        {Код клавиши}
    1: (CharCode: Byte;
       ScanCode: Byte));
evMessage: (                   {Событие-сообщение:}
Command: Word;                {Код команды}
  case Word of
    0: (InfoPtr: Pointer);
    1: (InfoLong: LongInt);
    2: (InfoWord: Word);
    3: (InfoInt: Integer);
    4: (InfoByte: Byte);
    5: (InfoChar: Char));
end;
```

Стандартная маска *evCommand* позволяет выделить из потока событий только те, которые связаны с передачей команд между различными обработчиками событий. Именно таким способом стандартный обработчик *TApplication.HandleEvent* сообщает новому обработчику *TNotebook.HandleEvent* о возникновении события, связанного с вновь определенной командой. Если бы мы не предусмотрели вызов стандартного обработчика с помощью оператора

```
Inherited HandleEvent(Event);
```

нам пришлось бы самим анализировать положение мыши или нажатую клавишу и интерпретировать их как соответствующие команды. Включение вызова *TApplication.HandleEvent* в тело нашего обработчика событий избавляет нас от этой рутинной работы.

В конце обработчика мы вызвали стандартную процедуру *ClearEvent*, с помощью которой в переменную *Event* помещается сообщение *Nothing* («пустое» событие). Это событие игнорируется всеми обработчиками, так что программа будет повторять проверку состояния мыши и клавиатуры до тех пор, пока не произойдет нового события. Фактически тело процедуры *TApplication.Run* (см. раздел исполняемых операторов нашей программы) состоит из бесконечно повторяющегося цикла проверки мыши и клавиатуры и передачи событий по цепи обработчиков событий. После получения любого события обработчик должен либо обработать это событие и очистить переменную *Event*, либо просто вернуть управление обработчику верхнего уровня, если эта команда не предназначена для него, либо, наконец, сформировать и передать новое событие для реализации команд, которые распознаны им, но которые он выполнять не умеет или не должен.

15.6. ПРОГРАММИРОВАНИЕ ДИАЛОГОВЫХ ЗАПРОСОВ

В обработчике событий *TNotebook.HandleEvent* мы предусмотрели вызовы нескольких процедур, с помощью которых реализуются конкретные действия программы. Настала пора запрограммировать эти действия.

Начнем с процедуры *FileOpen*. Ее задача — выбрать один из возможных файлов с данными и подготовить его к работе. Конечно, программу можно было бы сделать менее гибкой, раз и навсегда «привязав» ее к какому-то одному файлу, скажем, с именем *notebook.dat*. Но даже и в этом случае следует решить проблему с местоположением файла данных, а также определить, что должна делать программа, если нужный файл не найден. Наша программа будет весьма гибкой в этом отношении: она позволит указать интересующий нас файл мышью или клавишами курсора, либо ввести имя файла с помощью клавиатуры или взять его из буфера ранее введенных имен. Иными словами, поведение нашей программы будет в точности повторять поведение среды Турбо Паскаль в момент нажатия на клавишу F3.

Если Вы когда-либо программировали подобные действия в Турбо Паскале, Вы по достоинству оцените простоту их реализации в Turbo Vision:

Procedure FileOpen;

{Открывает файл данных}

var

PF: PFileDialog; {Диалоговое окно выбора файла}

Control: Word;

s: PathStr;

begin

{Создаем экземпляр динамического объекта:}

New(PF, Init('*.*dat', 'Выберите нужный файл:',
'Имя файла', fdOpenButton, 0));

{С помощью следующего оператора окно выводится на экран
и результат работы пользователя с ним помещается в
переменную Control:}

Control := DeskTop^.ExecView(PF);

{Анализируем результат запроса:}

case Control of

StdDlg.cmFileOpen, cmOk:

begin

PF^.GetFileName(s);

{.....}

end;

end; {case Control}

Dispose(PF, Done)

end; {FileOpen}

{Пользователь указал имя файла:}

{s содержит имя файла}

{Открыть файл}

{Уничтожаем экземпляр}

Для реализации этого фрагмента необходимо указать имя модуля *StdDlg* в предложении *Uses* — в этом модуле описан тип *PFileDialog* и предусмотрены все необходимые методы для работы с ним. Кроме того, в про-

грамме используется переменная *S* типа *PathStr*. Этот тип описан в модуле *DOS* — сошлитесь также и на него. Сделайте нужные изменения в тексте программы, не раскрывая пока сущности действий

{Открыть файл}

запустите программу на счет и нажмите клавишу *F3* — экран приобретет вид, показанный на рис.15.3.

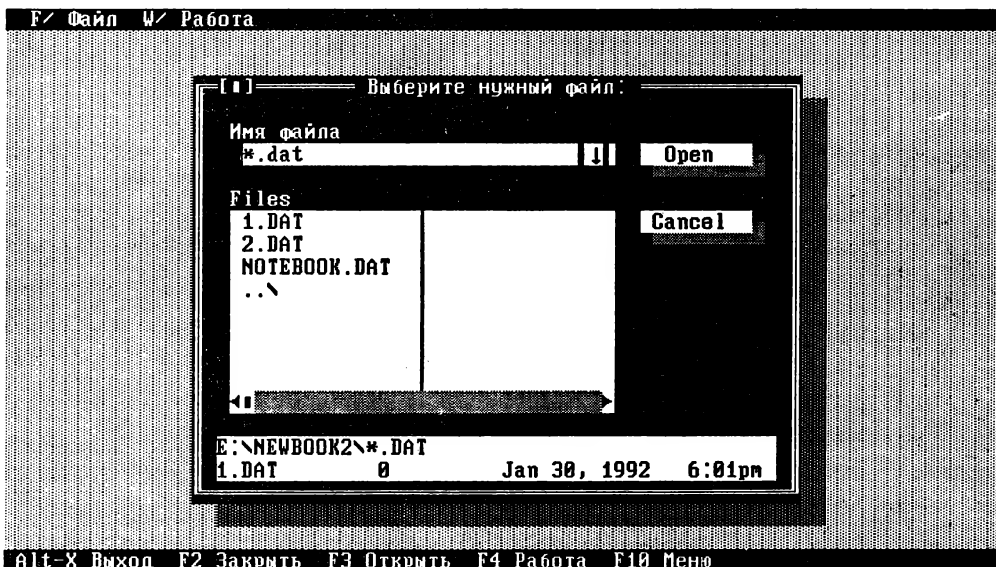


Рис.15.3. Диалоговое окно выбора файлов

Тип *PFileDialog* — это указатель на объект *TFileDialog*, создающий и обслуживающий стандартное диалоговое окно выбора файлов. Все действия по созданию и использованию диалогового окна, показанного на рис.15.3, реализуются двумя операторами:

```
New(PF, Init('*.dat', 'Выберите нужный файл:',
             'Имя файла', fdOpenButton, 0));
Control := DeskTop^.ExecView(PF);
```

Первый оператор инициирует новый экземпляр объекта *TFileDialog*. Три строковых параметра обращения к конструктору *Init* этого объекта задают, соответственно, маску выбираемых файлов ('*.dat'), заголовок диалогового окна ('Выберите нужный файл:') и заголовок окна ввода ('Имя файла'). Параметр *fdOpenButton* указывает на необходимость включить в диалоговое окно кнопку *Open*. Последним параметром задается идентификатор протокола ввода. Доступ к этому протоколу открывается кнопкой [↓] справа от окна ввода. Сам протокол хранится в куче в виде последовательности введившихся ранее текстовых строк. Идентификатор протокола ввода позволяет при необходимости использовать один и тот же протокол в разных диалоговых окнах.

Второй оператор

```
Control := DeskTop^.ExecView(PF);
```

помещает вновь созданное окно в основное поле экрана программы (ссылка *DeskTop^*) и инициирует диалог с пользователем. Результат диалога возвращается в переменной *Control*, значение этой переменной анализируется оператором

```
case Control of
.....
end;
```

Если *Control* содержит коды команд *cmOk* или *cmFileOpen*, то с помощью метода *GetFileName* объекта *TFileDialog* в переменную *S* записывается полное имя файла (с предшествующим путем)¹. Перед выходом из процедуры *FileOpen* экземпляр объекта *TFileDialog* уничтожается (удаляется из кучи) обращением к деструктору *Done*.

По описанной схеме в Turbo Vision создаются и используются любые другие диалоговые окна.

Для реализации диалогового запроса необходимо создать диалоговое окно и с помощью функции *ExecView* объекта-владельца (программы) инициировать диалог с пользователем. Результат, возвращаемый этой функцией, будет содержать выбранную пользователем команду.

Чтобы запрограммировать действия, связанные с открытием файла, следует вначале решить, какие именно данные он будет содержать. Напомню, что мы разрабатываем диалоговую программу управления «записной книжкой». Структура типичной записи в такой книжке состоит из трех полей: имя, телефон, адрес. Учитывая это, будем считать, что данные в файле хранятся в виде следующих записей:

```
const
  LName = 25;           {Длина поля Name}
  LPhone = 11;          {Длина поля Phone}
  LAddr = 40;           {длина поля Addr}
type
  DataType = record    {Тип данных в файле}
    Name : String[LName]; {Имя}
    Phone: String[LPhone]; {Телефон}
    Addr : String[LAddr];  {Адрес}
  end;
```

¹ В методе *TFileDialog.GetFileName* (**var** Name: PathStr) параметр обращения должен иметь тип PathStr. Этот тип определен в модуле DOS, — вот почему нам понадобилось сослаться на этот модуль в предложении *Uses*. Если указать компилятору на необходимость смягчить проверку строковых типов (директива компилятора *{SV-}*), то при обращении к *GetFileName* можно использовать переменную любого строкового типа, в том числе **String**.

Поместим эти строки в начале программы, а перед описанием процедуры *FileOpen* вставим определения следующих глобальных переменных:

var

```
DataFile: file of DataType; {Файловая переменная}
OpFileF : Boolean;           {Флаг открытого файла}
```

Дополним текст процедуры *FileOpen* такими строками:

case Control of

```
StdDlg.cmFileOpen, cmOk:
```

begin

```
PF^.GetFileName(s);
```

```
Assign(DataFile,s); {Отсюда начинаются новые строки}
```

```
{SI-}
```

```
Reset(DataFile);
```

```
if IOResult <> 0 then
```

```
  Rewrite(DataFile);
```

```
OpFileF := IOResult=0;
```

```
{SI+}
```

```
if OpFileF then
```

begin

```
  DisableCommands(WinCom2);
```

```
  EnableCommands(WinCom1)
```

end

end;

end;

С помощью оператора *DisableCommands* мы временно запрещаем набор команд, указанный в константе *WinCom2*. Эта константа в нашем случае должна содержать команду *cmOpen*; ее определение нужно включить сразу за определением константы *WinCom1*:

const

```
WinCom1: TCommandSet = [cmSave, cmWork];
```

```
WinCom2: TCommandSet = [cmOpen];
```

Обращение к процедуре *EnableCommands* разрешает использовать команды *cmSave* и *cmWork*.

15.7. ИНКАПСУЛЯЦИЯ НОВЫХ ПОЛЕЙ И МЕТОДОВ

При попытке откомпилировать полученный вариант программы Турбо Паскаль сообщит о неизвестном идентификаторе *DisableCommands*. На первый взгляд это кажется странным — ведь аналогичное обращение в обработчике событий *TNotebook.HandleEvent* не вызывало проблем! Все дело в том, что мы работаем с объектами, а следовательно, здесь очень важным становится контекст программ. Обработчик *TNotebook.HandleEvent* — это метод объекта *TNotebook*, который унаследовал от своих родителей многие свойства, в том числе и метод *DisableCommands*. Процедура

FileOpen не является потомком объектов Turbo Vision и не имеет доступа к их полям и методам.

В Turbo Vision все новые процедуры обычно инкапсулируются в объекты, если в них необходимо получить доступ к специфическим средствам этих объектов.

Поскольку процедура *FileOpen* вызывается из обработчика событий объекта *TNotebook*, нам следует включить ее в виде нового метода этого объекта:

```

type
  TNotebook = object (TApplication)
    .....
    Procedure FileOpen;
    Procedure FileSave;
    Procedure ChangeDir;
    Procedure DOSCall;
    Procedure Work;
  end;

```

В этом фрагменте мы инкапсулировали в объект все методы, используемые обработчиком событий. Разумеется, необходимо соответствующим образом изменить заголовок процедуры *FileOpen*, поскольку она теперь стала методом объекта *TNotebook*:

```

Procedure TNotebook.FileOpen;

```

Аналогичным образом следует изменить и заголовки других инкапсулированных процедур. Теперь трансляция пройдет успешно, а после открытия файла станет недоступна команда *F3*.

Тексты двух других новых методов объекта *TNotebook* не нуждаются в особых комментариях:

```

Procedure TNotebook.FileSave;
{Закрывает файл данных}
begin
  Close(DataFile);
  OpFileF := False;
  EnableCommands(WinCom2); {Разрешаем открыть файл}
  DisableCommands(WinCom1) {Запрещаем работу и сохранение}
end; {TNotebook.FileSave}
{-----}
Procedure TNotebook.ChangeDir;
{Изменяет текущий каталог}
var
  PD: PChDirDialog; {Диалоговое окно смены каталога/диска}
  Control: Word;
begin

```



```

New(PD, Init(cdNormal, 0));           {Создаем диалоговое окно}
Control := DeskTop^.ExecView(PD);    {Используем окно}
ChDir(PD^.DirInput^.Data^);          {Устанавливаем
                                     новый каталог}
Dispose(PD, Done)                     {Удаляем окно из кучи}
end; {TNotebook.ChangeDir}

```

Несколько слов по поводу реализации процедуры *TNotebook.ChangeDir*. В ней используется объект *TChDirDialog*, входящий в модуль *StdDlg*. С помощью этого объекта создается диалоговое окно, позволяющее выбрать новый диск или каталог. После создания и использования экземпляра объекта *TChDirDialog* в его поле

DirInput^.Data^

устанавливается строка типа *PathStr*, задающая новый каталог (и, возможно, новый диск).

Чуть сложнее обстоит дело с процедурой *DOSCall*, которая должна реализовать временный выход в ДОС. Дело в том, что перед выходом необходимо сохранить в куче текущее состояние программы, а после возврата нужно восстановить состояние программы, в том числе и вид экрана. Чтобы реализовать имеющиеся в Turbo Vision средства сохранения и восстановления программы, в предложение *Uses* необходимо добавить ссылку на модуль *Memory*. Вот текст метода *TNotebook.DOSCall*:

```

Procedure TNotebook.DOSCall;
{Временный выход в ДОС}
const
    txt = 'Для возврата введите EXIT в ответ'+
          ' на приглашение ДОС...';
begin
    DoneEvents;           {Закрыть обработчик событий}
    DoneVideo;            {Закрыть монитор экрана}
    DoneMemory;           {Закрыть монитор памяти}
    SetMemTop(HeapPtr);   {Освободить кучу}
    WriteLn(txt);         {Сообщить о выходе}
    SwapVectors;          {Установить стандартные векторы}
    {Передать управление командному процессору ДОС:}
    Exec(GetEnv('COMSPEC'), '');
    {Вернуться из ДОС:}
    SwapVectors;          {Восстановить векторы}
    SetMemTop(HeapEnd);   {Восстановить кучу}
    InitMemory;           {Открыть монитор памяти}
    InitVideo;            {Открыть монитор экрана}
    InitEvents;           {Открыть обработчик событий}
    InitSysError;         {Открыть обработчик ошибок}
    Redraw                {Восстановить вид экрана}
end; {DOSCall}

```

Процедуры *DoneXXXX* завершают работу отдельных частей Turbo Vision, а процедуры *InitXXXX* осуществляют обратные действия. С помощью процедуры *SetMemTop* в ДОС передается информация о фактически используемой динамической памяти (по умолчанию программе предоставляется вся доступная память). Этот вызов освобождает неиспользуемую в данный момент часть кучи для размещения в ней командного процессора *COMMAND.COM*. После возврата из ДОС вызов *SetMemTop* используется еще раз — для того, чтобы зарезервировать за программой всю ранее выделенную ей память. Процедура *Redraw* восстанавливает все видимые элементы экрана.

15.8. СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ ГРУПП

Пора заняться основной содержательной частью нашей программы — процедурой *Work*. Прежде всего следует продумать способ взаимодействия пользователя с данными (интерфейс пользователя). От удачного выбора интерфейса во многом зависит успех разработки диалоговых программ: неудобный способ доступа, связанный с необходимостью ввода каких-либо команд или ответов на многочисленные вопросы, надолго отобьет у пользователя всякое желание работать с программой. Turbo Vision предоставляет в Ваше распоряжение все необходимые средства для разработки современного объектно-ориентированного диалога. В ходе такого диалога пользователь видит на экране объекты, о которых идет речь, он может указать на любой объект и выбрать те действия, которые нужно осуществить над ним.

При работе с электронной записной книжкой хотелось бы, чтобы на экране появилось сразу несколько записей, отсортированных в алфавитном порядке. Пользователь должен иметь возможность «листать» книжку, отыскивать в ней нужную запись, добавлять новые и исключать ненужные записи, редактировать их (вносить изменения). Таким образом, ядром диалога должно стать окно с текстом. При необходимости пользователь может смещать текст в окне в ту или иную сторону, перемещать само окно относительно границ экрана, менять его размеры. Все эти возможности типичны для многочисленных текстовых редакторов, систем программирования, систем управления базами данных и т.п.

Для реализации этих действий в Turbo Vision предусмотрен специальный объект *TWindow*, экземпляры которого отображаются на экране в виде прямоугольного окна с рамкой и стандартными кнопками изменения размера и закрытия окна. Попробуем создать такое окно в нашей программе. Для этого изменим текст процедуры *Work* следующим образом:

```
Procedure TNotebook.Work;  
  {Работа с данными}  
var  
  R: TRect;  
begin  
  R.Assign(0,0,80,23);  
  Desktop^.Insert(New(PWindow, Init(R,'',0)),  
end; {Work}
```

После запуска программы нажмите клавишу *F3*, укажите в диалоговом окне имя несуществующего файла (файл данных пока еще не создан), нажмите клавиши *Enter* и *F4* — экран приобретет вид, показанный на рис.15.4.

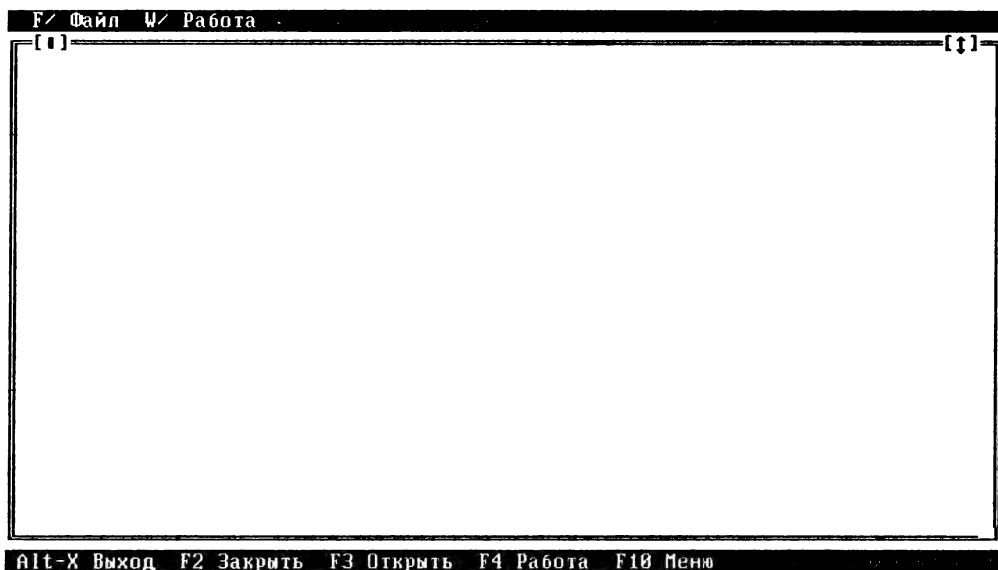


Рис.15.4. Окно просмотра данных

Если Ваш ПК оснащен устройством ввода типа мышь, Вы можете перемещать это окно по экрану (надо «схватить» мышью верхнюю рамку окна, т.е. подвести к ней указатель мыши, нажать левую кнопку и, удерживая кнопку нажатой, перемещать мышь), изменять его размеры («схватить» правый нижний угол), использовать стандартные кнопки изменения размера (справа на верхней рамке) и закрытия окна (слева). Ничего другого окно не умеет. А как загрузить в него текст? Как получить хорошо знакомые по среде Турбо Паскаль полосы — указатели и управлять с их помощью положением текста? Для этих целей можно было бы использовать объект *TScroller*, представляющий собой окно с текстом и с двумя полосами — указателями. Однако по умолчанию такое окно не имеет рамки, а потому не может изменять своего размера, в нем нет стандартных кнопок изменения размера и закрытия окна. Таким образом, и объект *TScroller* не решает всех проблем. Каков же выход? Нужно создать новый объект, объединяющий в себе свойства и *TWindow*, и *TScroller*! В терминах Turbo Vision такие составные объекты называются *группами*.

Введем в программу следующий объект:

```
type
  PWorkWin = ^TWorkWin;
  TWorkWin = object (TWindow)
    Constructor Init(Bounds: TRect);
  end;
```

Новый объект является потомком *TWindow* и, следовательно, наследует все свойства родителя, в том числе рамку и способность перемещения по экрану. Дополнительные свойства ему должен придать новый конструктор *TWorkWin.Init*, которому мы в качестве параметра передаем начальное положение и размеры создаваемого окна:

```
Constructor TWorkWin.Init(Bounds: TRect);
{Создание окна данных}
var
    HS,VS: PScrollBar;           {Полосы-указатели}
    Interior: PScroller;         {Указатель на
                                управляемое текстовое окно}
begin
    TWindow.Init(Bounds, '', 0); {Создаем новое окно с рамкой}
    GetClipRect(Bounds);        {Получаем в BOUNDS координаты
                                минимальной перерисовываемой
                                части окна}
    Bounds.Grow(-1,-1);         {Устанавливаем размеры
                                окна с текстом}
    {Включаем стандартные по размеру и положению
    полосы-указатели:}
    VS := StandardScrollBar(sbVertical+sbHandleKeyBoard);
    HS := StandardScrollBar(sbHorizontal+sbHandleKeyBoard);
    {Создаем текстовое окно:}
    Interior := New(PScroller, Init(Bounds, HS, VS));
    Insert(Interior) {Включаем его в основное окно}
end; {TWorkWin.Init}
```

С помощью вызова процедуры *GetClipRect* мы получаем размеры минимального прямоугольника, который следует обновлять при любых перемещениях окна или изменениях его размера. Такой вызов позволяет до минимума сократить время вывода. Процедура *Bounds.Grow* изменяет вертикальный и горизонтальный размеры прямоугольника *Bounds*: при положительном параметре соответствующий размер увеличивается, при отрицательном — уменьшается. Параметры *-1, -1* учитывают рамку основного окна. Функция *StandardScrollBar* создает указатель на управляющую полосу стандартного размера. При обращении к ней параметр *sbVertical* (*sbHorizontal*) определяет положение полосы, а параметр *sbHandleKeyboard* разрешает использование клавиатуры для управления ею (если этот параметр не включить, полоса будет управляться только с помощью мыши). Наконец, процедура *Insert* включает вновь созданное окно *TScrollBar* в основное окно *TWindow*, так что теперь оба окна будут функционировать как одно целое.

Для создания группы необходимо в объект-потомок от *TGroup* (обычно — это объект *TWindow*)

**или потомок от него) вставлять нужные элементы
с помощью метода *Insert*.**

Осталось лишь нужным образом изменить процедуру *Work*:

```
Procedure TNotebook.Work;  
  {Работа с данными}  
var  
  R: TRect;  
  PW: PWorkWin;  
begin  
  R.Assign(0,0,80,23);  
  PW := New(PWorkWin, Init(R));  
  DeskTop^.Insert(PW)  
end; {Work}
```

Если исполнить подготовленную таким образом программу, на экране появится изображение, показанное на рис.15.5.

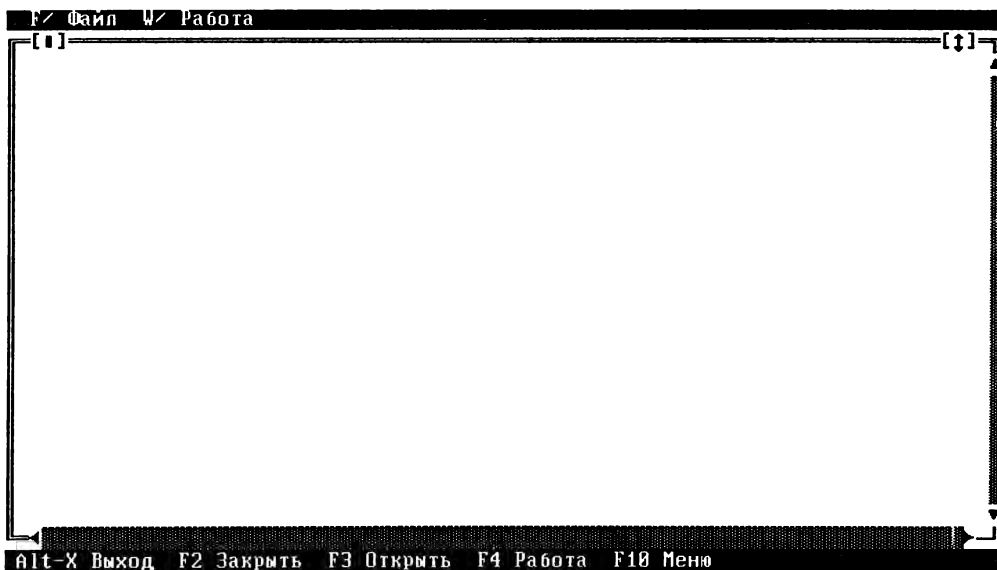


Рис.15.5. Окно с полосами прокрутки

15.9. ВЫВОД ТЕКСТА

По сравнению с рис.15.4 мы добились немногого, ведь пока еще не решена главная проблема — вывод нужного текста. Разумеется, в Вашем распоряжении всегда имеется процедура *WRITELN*, однако вывод текста «в лоб» с помощью этой процедуры практически никогда не используется в Turbo Vision, так как в этом случае выведенный текст не будет связан с окнами.

В объекте *TScroller* для вывода текста предусмотрен абстрактный метод *Draw*. Абстрактным он называется потому, что не выполняет никакой

полезной работы. Однако именно к этому методу обращается обработчик событий объекта *TScroller* всякий раз, когда понадобится обновить на экране вид окна. Чтобы объект выполнял все заложенные в него функции, нам необходимо перекрыть этот метод новым. Мы уже знаем, что для этого нужно объявить новый объект:

```
type
  PInterior = ^TInterior;
  TInterior = object (TScroller)
    Constructor Init(var Bounds: TRect;
                    HS,VS: PScrollBar);
    Procedure Draw; Virtual;
    Procedure ReadFile;
  end;
```

Мы перекрыли абстрактный метод *Draw*, стандартный конструктор *Init* и инкапсулировали в объект новый метод *ReadFile*. Новый конструктор предназначен для инициации экземпляра объекта *TScroller*. Кроме того, с помощью метода *ReadFile* он должен прочитать все записи файла данных и подготовить соответствующий массив строк — это сократит время на обновление текста процедурой *Draw*.

Перед тем, как двигаться дальше, подумаем о способе хранения строк для процедуры *Draw*. Если все необходимые действия по чтению нужной записи из файла и преобразования ее к текстовому формату возложить на процедуру *Draw*, наша программа станет слишком медленной, в особенности, если файл данных записан на дискете. Поэтому предусмотрим такие глобальные переменные:

```
const
  MaxLine = 300;                      {Максимальная длина массива}
  LLine = LName+LPhone+LAddr; {Длина строки}
var
  NLines: Word;                       {Истинная длина массива строк}
  Lines: array [1..MaxLine] of String [LLine]; {Массив строк}
```

Теперь нетрудно подготовить процедуру *ReadFile*:

```
Procedure TInterior.ReadFile;
{Читает содержимое файла данных в массив Lines}
var
  k: Integer;
  s: String;
  Data: DataType;
begin
  seek(DataFile,0);
  NLines := FileSize(DataFile);
  if NLines > MaxLine then
    NLines := MaxLine;
  for k := 1 to NLines do
    begin
```

```

Read(DataFile, data);
with data do
  begin
    s := Name;
    while Length(s) < LName do
      s := s+' ';
    s := s+Phone;
    while Length(s) < LName+LPhone do
      s := s+' ';
    s := s+Addr
  end;
  Lines[k] := s
end;
end; {ReadFile}

```

В этой процедуре из записей файла данных готовится массив строк *Lines*, причем начало каждого поля выравнивается так, чтобы поля образовали колонки — такая форма вывода поможет легко найти на экране каждое поле.

Теперь займемся процедурой *Draw*:

```

Procedure TInterior.Draw;
{Выводит данные в окно просмотра}
var
  n,                {Текущая строка экрана}
  k: Integer;       {Текущая строка массива}
  B: TDrawBuffer;
  Color: Byte;
begin
  Color := GetColor(1); {Использовать цвет
                        основного текста}
  for n := 0 to pred(Size.Y) do
    {Size.Y - количество строк окна}
    begin
      k := Delta.Y+n+1; {Delta.Y - номер
                        первой выводимой строки}
      MoveChar(B, ' ', Color, Size.X);
      MoveStr(B,
        Copy(Lines[k], Delta.X+1, Size.X), Color);
      WriteLine(0, N, Size.X, 1, B)
    end
  end; {TInterior.Draw}

```

Работа процедуры основана на использовании текущих размеров и положения текстового окна относительно текста. Эти параметры хранятся в полях *Size* и *Delta* объекта *TScroller* и обновляются всякий раз, когда пользователь манипулирует полосами управления или изменяет размеры окна. Для вывода текста используются три процедуры: *MoveChar*, *MoveStr*, *WriteLine*. Каждая из них оперирует переменной *B* типа *TDrawBuffer*,

представляющей собой последовательности кодов выводимых символов и их атрибутов. Процедура *MoveChar* заполняет переменную *B* указанным символом (' ') и атрибутом (*Color*). Процедура *MoveStr* копирует строку в переменную *B*, а с помощью *WriteLine* осуществляется вывод буфера *B* на экран.

Для вывода изображений (текста) перекрывайте и используйте метод *Draw* объекта-владельца нужной части экрана. Это обеспечит автоматическое изменение изображения и его прорисовку при изменении границ или положения поля вывода.

15.10. ЦВЕТОВАЯ ПАЛИТРА

В процедуре *Draw* переменная *Color* задает атрибуты (цвет символов и цвет фона) символов, выводимых с помощью методов *MoveChar* и *MoveStr*. С помощью функции *GetColor* она устанавливается таким образом, чтобы символы на экране отображались цветовым сочетанием с номером 1. В Turbo Vision используется гибкая система установки цвета отдельных видимых элементов. Элемент изображения связывается не с каким-то конкретным цветом, а с индексом в таблице цветов, называемой палитрой. Количество элементов палитры зависит от количества цветовых сочетаний, используемых при выводе элемента изображения. Например, в *TScroller* используется двухэлементная палитра цветов: первый элемент устанавливает цвет нормального текста, второй — выделенного текста (рис.15.6).

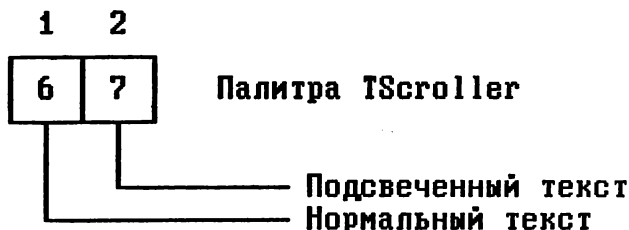


Рис.15.6. Палитра объекта *TScroller*

Числа 6 и 7 в этой палитре указывают не конкретные цвета, а номера позиций в палитре объекта-владельца. Для нашего случая объектом-владельцем будет *TWindow*. Таким образом, цвет номер 1 палитры *TScroller* лишь указывает на шестое по счету цветовое сочетание в палитре *TWindow* (рис.15.7).



Рис.15.7. Связь палитр TScroller и TWindow

Шестой элемент палитры *TWindow* в свою очередь ссылается на 13-й элемент палитры своего владельца — *TProgram*. Объект *TProgram* — это начальный видимый элемент любой программы в Turbo Vision. На нем заканчивается любая цепочка ссылок, т.е. его палитра содержит конкретные атрибуты символов. Тринадцатый элемент этой палитры содержит значение \$1E, что соответствует выводу желтого символа на синем фоне — именно таким образом отображается нормальный текст в окне *TScroller*, если это окно вставлено в *TWindow*. Если бы объект *TScroller* был помещен непосредственно на панель экрана, то значение 6 в первом элементе палитры *TScroller* указывало бы на 6-й элемент палитры *TProgram*, содержащий атрибуты \$28 (темно-серые символы на зеленом фоне). Цветовые палитры в Turbo Vision содержат такие значения по умолчанию, чтобы любая комбинация цветов давала приятную цветовую гамму. При необходимости пользователь может изменить любую цветовую палитру.

Вернемся к нашему примеру и рассмотрим реализацию конструктора *Init*:

```

Constructor TInterior.Init(var Bounds: TRect;
                           HS,VS: PScrollBar);
{Создает окно для данных}
begin
  Inherited Init(Bounds, HS, VS);
  ReadFile;
  GrowMode := gfGrowHiX + gfGrowHiY;
  SetLimit(LLine,NLines)
end; {TInterior.Init}

```

Объект *TScroller* имеет поле *GrowMode*, которое определяет, как элемент будет изменять свои размеры, если пользователь потребует этого. Параметр *gfGrowHiX* предписывает окну *TScroller* изменяться таким образом, чтобы правая его граница всегда находилась на постоянном расстоянии от правой границы владельца. Точно также *gfGrowHiY* задает неизменным расстояние нижней границы окна *TScroller* от нижней границы владельца. Таким образом, окно *TScroller* всегда будет занимать всю внутреннюю часть окна-владельца *TWindow*. С помощью процедуры *SetLimit (X,Y)* мы задаем горизонтальную *X* и вертикальную *Y* границы перемещения окна относительно текста. Эти границы будут выдерживаться при управлении окном с помощью клавишей или мыши: какими бы не были текущие размеры окна, нажатие на клавишу *End*, например, смещает его вправо так, чтобы самым правым видимым символом был *X*-й символ текста. Нажатие на клавиши *Ctrl-PgDn* смещает окно вниз по тексту таким образом, чтобы самая нижняя строка окна соответствовала *Y*-й строке текста. Иными словами, параметры *X* и *Y* задают координаты правого нижнего угла виртуального (воображаемого) экрана неограниченных размеров, на котором находится текст и по которому «скользит» окно. Левый верхний угол виртуального экрана всегда имеет координаты (0,0).

Осталось отредактировать конструктор *TWorkWin.Init*: нужно изменить тип переменной *Interior*

```
var
.....
Interior: PInterior;
```

и обращение к конструктору:

```
Interior := New(PInterior, Init(Bounds, HS, VS));
```

Но не спешите запускать программу на счет: ведь файла данных пока еще нет, а поэтому Вы ничего не увидите на экране. Чтобы все-таки оценить достигнутые результаты, измените текст процедуры *ReadFile* — добавьте в него следующие строки:

```
Procedure TInterior.ReadFile;
{Читает содержимое файла данных}
var
.....
f: text;
begin
s := copy(ParamStr(0),1,pos('.',ParamStr(0))+'pas';
assign(f,s);
reset(f);           {Открываем файл с текстом программы}
NLines := 0;
while not EOF(f) and (NLines < MaxLine) do
begin
inc(NLines);
ReadLn(f,Lines[NLines])
end;
```

```

close(f);
exit;
.....
end; {ReadFile}

```

Добавленные строки заставят процедуру прочитать в массив *Lines* текст самой программы (если Вы будете запускать программу из среды Турбо Паскаль, не забудьте установить компиляцию в дисковый файл опцией *COMPILE/DESTINATION*, иначе оператор

```
s:=copy(ParamStr(0),1,pos('.',ParamStr(0))+'pas');
```

не сможет установить в *S* правильное имя файла с текстом Вашей программы). После запуска программы нажмите клавишу *F3*, задайте имя несуществующего файла, нажмите клавиши *Enter* и *F4* — на экране появится изображение, показанное на рис.15.8.

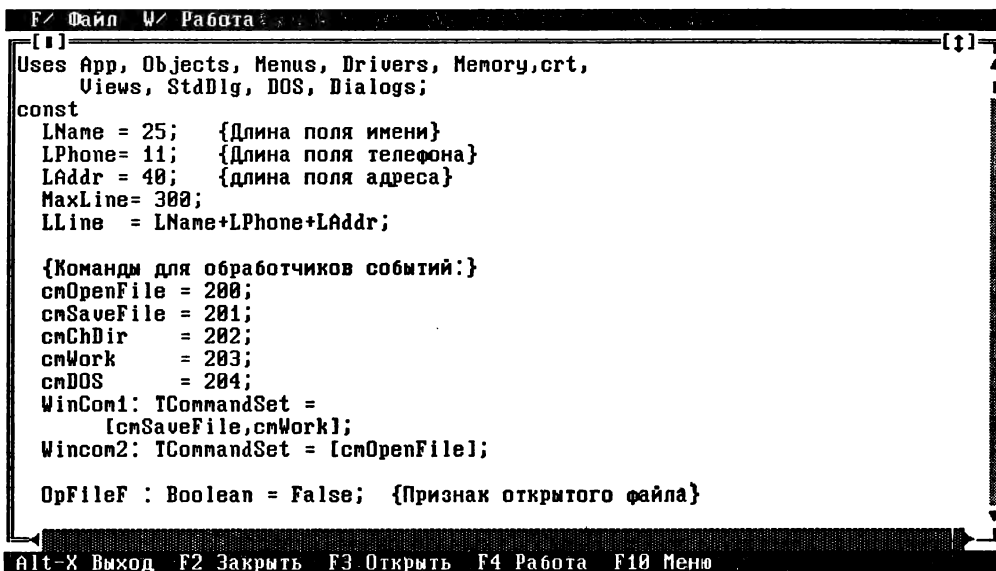


Рис.15.8. Окно с текстом программы.

Это окно откликается на нажатие клавиш управления курсором, команды *PgUp*, *PgDn*, *Ctrl-PgUp* и т.д. подобно тому, как ведет себя окно редактора в среде Турбо Паскаль. С помощью мыши Вы можете перемещать его по экрану, изменять размеры, закрывать — все эти действия реализует стандартный обработчик событий объекта *TScroller*.

15.11. ИСПОЛЬЗОВАНИЕ КОЛЛЕКЦИЙ

Для вывода текста мы использовали глобальный массив *Lines*. Как известно, длина любого массива в Турбо Паскале не может превышать длину сегмента данных (64 Кбайт). Это ограничение можно убрать, если воспользоваться еще одним механизмом Turbo Vision — коллекциями. Подобно массивам, коллекции представляют собой набор элементов, в

которых можно хранить любые данные, включая экземпляры любых объектов. К элементам коллекции можно обращаться по индексу, однако, в отличие от массива, коллекция размещается в куче, поэтому ее суммарная длина ограничивается всей доступной памятью и может быть больше 64 Кбайт. Кроме того, размер коллекции не лимитируется при ее создании и может динамически изменяться в процессе работы программы.

Коллекции обладают целым рядом новых свойств. В частности, к любой коллекции можно применить метод *ForEach*, который осуществит заданные Вами действия над каждым элементом коллекции. Таким способом можно, например, быстро отыскать элемент, удовлетворяющий заданным требованиям. Наконец, в Turbo Vision определены отсортированные коллекции, элементы которых упорядочиваются по заданному ключу. Все это делает коллекции более предпочтительным способом хранения данных, чем массивы Турбо Паскаля.

Попробуем заменить массив *Lines* на отсортированную коллекцию. Введем в объект *TInterior* новое поле *PS*:

```

type
  TInterior = object (TScroller)
    PS: PStringCollection;
    .....
  end;
```

Тип *PStringCollection* в Turbo Vision определен как указатель на экземпляр объекта *TStringCollection*, представляющий собой отсортированную коллекцию строк. Сортировка строк осуществляется по обычным правилам сравнения строк по ASCII-кодам. Если вновь помещаемая строка уже существует в коллекции, она не дублируется (при желании программист может разрешить дублирование одинаковых строк), поэтому в общем случае количество элементов коллекции может оказаться меньшим количества помещенных в нее строк.

Для создания коллекции удалите ненужные теперь глобальные объявления *MaxLine*, *Lines* и *NLines* (в коллекции есть другие средства доступа к элементам) и измените метод *ReadFile* следующим образом:

```

Procedure TInterior.ReadFile;
var
  .....
begin
  PS := New(PStringCollection, Init(100,10));
  s := copy(ParamStr(0),1,pos('.',ParamStr(0))+'pas';
  assign(f,s);
  reset(f); {Открыть файл с текстом программы}
  while not (EOF(f) or LowMemory) do
    begin
      ReadLn(f,s);
      if s <> ' ' then PS^.Insert(NewStr(s))
    end;
  Close(f);
```

```

exit;
Seek(DataFile,0);
while not (EOF(DataFile) or LowMemory) do
begin
  Read(DataFile, data);
  with data do
  begin
    .....
  end;
  if s<>' ' then PS^.Insert(NewStr(s))
end;
end; {ReadFile}

```

В приведенном фрагменте мы предусмотрительно изменили только ту часть программы, которая стоит после оператора *Exit* и которая зависит от удаленных глобальных определений. Вы должны сделать эти изменения (они все равно нам пригодятся) или закомментировать эту часть текста, чтобы получить синтаксически правильный вариант программы.

С помощью оператора

```
PS := New(PStringCollection, Init(100,10));
```

иницируется экземпляр коллекции, причем параметр 100 определяет начальный размер коллекции, а параметр 10 — шаг наращивания коллекции, если ее размер превысит 100 элементов. Оператор

```
if s<>' ' then PS^.Insert(NewStr(s))
```

вставляет очередную непустую строку в коллекцию. Заметим, что коллекция *PS* передается не строка *S*, а лишь указатель на нее, т.к. функция *NewStr* размещает строку в куче и возвращает ее адрес. Функция *NewStr* не может разместить в куче пустую строку, поэтому мы вставляем в коллекцию только непустые строки.

Функция *LowMemory* используется для контроля за размерами динамической памяти: она возвращает значение *True*, если в куче осталось менее 4 Кбайт.

В последний оператор метода *Interior.Init* внесите следующее изменение:

```

Constructor TInterior.Init(var Bounds: TRect;
                           HS,VS: PScrollBar);
begin
  .....
  SetLimit(LLine,PS^.Count)
end; {TInterior.Init}

```

Другим станет также и реализация метода *TInterior.Draw*:

```

Procedure TInterior.Draw;
var
  n,k: Integer;
  B: TDrawBuffer;

```

```

p: PString;
Color: Byte;
begin
  Color := GetColor(1);
  for n := 0 to pred(Size.Y) do
    begin
      k := Delta.Y+n;
      MoveChar(B, ' ', Color, Size.X);
      if k < pred(PS^.Count) then
        begin
          p := PS^.At(k);
          MoveStr(B, Copy(p^, Delta.X+1, Size.X), Color)
        end;
      WriteLine(0, N, Size.X, 1, B)
    end
  end; {TInterior.Draw}

```

Элементы коллекции нумеруются, начиная с номера 0. Длина коллекции (общее количество ее элементов) хранится в поле `PS^.Count`. Функция `PS^.At(k)` возвращает указатель на k -й элемент коллекции.

Созданная коллекция размещается в динамической памяти, поэтому после использования ее следует удалить из кучи. Для этого перекроем стандартный деструктор *Done*:

```

type
  TInterior = object (TScroller)
    .....
    Destructor Done; Virtual;
  end;
Destructor TInterior.Done;
begin
  Dispose(PS, Done);      {Удаляем коллекцию}
  Inherited Done          {Выполняем стандартный деструктор}
end;

```

Еще раз хочу обратить Ваше внимание на особенность программирования в среде Turbo Vision: Вы определяете метод, но не указываете, когда он должен быть выполнен. Правильно сконструированный объект уже «знает», когда он ему понадобится! Так было в случае правила *Draw*, так же обстоит дело и с деструктором *Done*: обработчик событий окна *TWindow* вызовет этот метод, как только он получит событие *stCancel* (закрыть окно). Чтобы убедиться в этом, установите контрольную точку в строке

`Dispose(PS, Done); {Удаляем коллекцию}`

и запустите программу. Останов в контрольной точке произойдет только в том случае, если Вы загрузите окно с текстом и попытаетесь выйти из программы. Если из программы выйти сразу после ее запуска, контрольная точка не работает.

Вид экрана с окном просмотра отсортированного файла показан на рис.15.9.

```

F/ Файл  W/ Работа
[[ ]]----- Выберите нужную запись. ESC - переход к диалогу -----[{}]-
      располагаться в теле обработчика событий
                        диапазон контекстной справочной
                        службы}
                                статуса Menus}
                                fdOpenButton,0));
                        HS,US: PScrollBar);
      DisableCommands(WinCom2)
      EnableCommands(WinCom1);
FileOpen;
NIL)))))) {Нет других элементов подменю}
NewItem('A/ Открыть','F3',kfF3,cmOpenFile,hcNoContext,
NewItem('B/ Закрыть','F2',kfF2,cmSaveFile,hcNoContext,
NewItem('C/ Сменить диск','',0,cmChDir,hcNoContext,
NewItem('D/ Вызов ДОС','',0,cmDOS,hcNoContext,
NewItem('D/ Конец работы','Alt-X',kbAltX,cmQuit,hcNoContext,
NewLine(
Rewrite(DataFile);
begin
end
if OpFileF then Work
s := Name;
Alt-X Выход  F2 Закрыть  F3 Открыть  F4 Работа  F10 Меню

```

Рис.15.9. Окно с отсортированным тестом программы

15.12. УКАЗАТЕЛЬ НА ЭЛЕМЕНТ СПИСКА

Как уже отмечалось, с помощью процедуры *Draw* можно выводить обычный текст и выделенный текст. Попробуем использовать это обстоятельство для того, чтобы поместить в окно просмотра указатель на текущий элемент данных. Для этого добавим в *TInterior* еще одно поле:

```

type
  TInterior = object (TScroller)
    Location: Word;
    .....
  end;

```

Поле *Location* будет хранить номер той строки, которая отождествляется с выбранной строкой и которая на экране должна выделяться цветом. Добавьте в конце метода *ReadFile* строку

```
Location := 0;
```

и измените метод *Draw*:

```

Procedure TInterior.Draw;
{Выводит данные в окно просмотра}
var
  n,k: Integer;
  B: TDrawBuffer;
  p: PString;

```

```

Color: Byte;
begin
  if Delta.Y > Location then
    Location := Delta.Y;
  if Location > Delta.Y+pred(Size.Y) then
    Location := Delta.Y+pred(Size.Y);
  for n := 0 to pred(Size.Y) do
    begin
      k := Delta.Y+n;
      if k=Location then
        Color := GetColor(2)
      else
        Color := GetColor(1);
      .....
    end
  end; {TInterior.Draw}

```

Вначале проверяется, попадает ли строка с номером, хранящимся в *Location*, в число выводимых строк. Если это не так, значит пользователь изменил размеры окна или сдвинул его относительно текста; в этом случае нужным образом корректируется значение *Location*. Такая проверка гарантирует, что в окне всегда будет выводиться текущая строка. Перед выводом очередной строки сравнивается значение ее номера с величиной *Location* и, если величины совпадают, строка выводится цветом 2 из палитры *TScroller* (темно-синими символами на сером фоне).

Создав указатель в окне, нужно предусмотреть и средства воздействия на него. Для этого нам понадобится проверять действия пользователя с мышью и клавиатурой и изменять положение указателя. Вы не забыли, что все действия программы в Turbo Vision выполняются с помощью обработчика событий? Перекроем стандартный метод *HandleEvent* в объекте *TInterior*:

```

type
  TInterior = objet(TScroller)
  .....
  Procedure HandleEvent(var Event: TEvent); Virtual;
end;
Procedure TInterior.HandleEvent(var Event: TEvent);
{Обработчик событий для окна данных}
var
  R: TPoint;
begin
  Inherited HandleEvent(Event);
  case Event.What of
    evMouseDown: {Реакция на щелчок мышки}
      begin
        MakeLocal(MouseWhere, R); {Получаем в R локальные
                                   координаты указателя мыши}
        Location := Delta.Y+R.Y;

```



```

    Draw
  end;
  evKeyDown: {Реакция на клавиши + -}
  case Event.KeyCode of
    kbGrayMinus: if Location > Delta.Y then
      begin
        dec(Location);
        Draw
      end;
    kbGrayPlus: if Location < Delta.Y+pred(Size.Y) then
      begin
        inc(Location);
        Draw
      end;
  end
end
end; {TInterior.HandleEvent}

```

В новом методе вначале вызывается унаследованный обработчик событий *TScroller.HandleEvent*, с помощью которого обрабатываются все стандартные действия с окном (смещение текста, изменение размеров и т.д.). Затем обрабатываются события от нажатия кнопки мыши и от нажатия клавиш «+» и «-» из зоны цифровых клавиш (на клавиатуре ПК они выделяются серым цветом). С клавишей «+» связывается действие «Сместить указатель вниз на одну строку», с клавишей «-» — «Сместить вверх». Выбор серых клавиш «+» и «-» для смещения указателя вызван тем, что клавиши управления курсором используются для смещения окна и обрабатываются стандартным обработчиком событий. Заметим, что нажатие кнопки мыши будет обрабатываться в *TScroller.HandleEvent* только в том случае, если указатель мыши находится на рамке окна или на полосах управления. Если указатель сместить внутрь окна, нажатие на кнопку мыши будет преобразовано в событие *evMouseDone* и передано в наш обработчик. В этом случае глобальная переменная *MouseWhere* содержит абсолютные координаты указателя мыши (т.е. координаты относительно левого верхнего угла экрана). Чтобы получить номер соответствующей строки текста, мы сначала с помощью оператора

```
MakeLocal(MouseWhere, R);
```

получаем в переменной *R* локальные координаты мыши относительно границ окна *TScroller*. Оператор

```
Location := Delta.Y+R.Y;
```

устанавливает в поле *Location* номер той строки текста, на которой располагается указатель мыши.

15.13. ДИАЛоговое ОКНО ВЫБОРА РЕЖИМА

Подведем некоторые итоги. Мы создали программу, которая погружает пользователя в среду объектно-ориентированного диалога Turbo Vision: она поддерживает командные клавиши, работу с мышью, может сменить каталог или диск, выбрать нужный файл и загрузить его в окно просмотра. Не так плохо для 300 строк программного текста! Наша дальнейшая задача — реализовать другие режимы работы (поиск нужной строки, добавление и уничтожение строк, их изменение). Для двух из них (уничтожение и редактирование строки) в программе необходимо каким-то образом указать ту строку, с которой будет работать пользователь. Мы уже реализовали эту возможность, предусмотрев в окне просмотра текста управляемый указатель. Поэтому режим просмотра можно принять в качестве основного режима работы с данными. В связи с этим следует несколько изменить метод *TNotebook.HandleEvent*, предусмотрев в нем автоматический переход в режим просмотра данных в случае успешного открытия файла с данными:

```
Procedure TNotebook.HandleEvent(var Event: TEvent);
{Обработчик событий программы}
begin
  Inherited HandleEvent(Event);
  if Event.What = evCommand then
    case Event.Command of
      cmOpenFile:
        begin
          FileOpen;
          if OpFileF then Work
        end;
      .....
    end; {TNotebook.HandleEvent}
```

Как из режима просмотра данных перейти к другим режимам? Возможно несколько решений. Я предлагаю для этих целей воспользоваться командой *cmClose* (закрыть окно просмотра): в момент, когда пользователь в режиме просмотра данных нажмет клавишу *Esc* или воздействует мышью на кнопку «Закрыть окно», на экране должно раскрыться диалоговое окно выбора режима, предлагающее одно из пяти возможных продолжений:

- закрыть окно просмотра;
- удалить текущую запись;
- искать нужную запись;
- редактировать текущую запись;
- добавить запись (записи).

Для реализации этой идеи в уже созданный нами обработчик событий *TInterior.HandleEvent* следует ввести обработку события *cmClose*:

```
const
{Команды для обработчиков событий:}
.....
```

```
cmAdd : AddItem(False);
```

```

end
end;
cmZoom: exit;
end;
evMouseDown: {Позиционировать мышью}
.....
evKeyDown: {Позиционировать клавишами + -}
case Event.KeyCode of
kbEsc: goto Cls;
kbGrayMinus: if Location > Delta.Y then
.....
end; {TInterior.HandleEvent}

```

В этом фрагменте мы расширили набор нестандартных команд (константы *cmCan*, ..., *cmAdd*), ввели новую функцию *Control* и предусмотрели необходимые процедуры в теле обработчика событий. Заметим, что режимы редактирования записи и добавления новой записи очень схожи по организации диалога с пользователем, поэтому они реализуются в рамках одной процедуры *AddItem* и управляются параметром обращения к ней.

Функция *Control* используется для создания диалогового окна выбора продолжения. В качестве значения этой функции будет возвращаться одна из пяти новых команд. В начальном варианте функция возвращает команду *cmCan*, что интерпретируется обработчиком событий как указание на завершение работы с диалоговым окном. Поэтому, если Вы вставите указанный текст в программу и запустите ее, поведение программы останется прежним.

Займемся реализацией функции *Control*. Она должна создать диалоговое окно выбора режима, получить с его помощью команду, идентифицирующую выбранный режим, и вернуть эту команду в качестве своего значения:

```

Function Control: Word;
{Получает команду из основного диалогового окна}
const
  X = 1;
  L = 12;
  DX = 13;
  But: array [0..4] of String [13] = {Надписи на кнопках:}
  ('~1~ Выход ', '~2~ Убрать ', '~3~ Искать ',
  '~4~ Изменить ', '~5~ Добавить ');
  Txt: array [0..3] of String [52] = (
  {Справочный текст:}
  'Убрать - удалить запись, выделенную цветом',
  'Искать - искать запись, начинающуюся нужными буквами',
  'Изменить - изменить поле (поля) выделенной записи',
  'Добавить - добавить новую запись');
var
  R: TRect;

```

```

D: PDialog;
k: Integer;
begin
  R.Assign(7,6,74,15);
  D := New(PDialog,Init(R, 'Выберите продолжение:'));
  with D^ do
    begin
      for k := 0 to 3 do {Вставляем поясняющий текст}
        begin
          R.Assign(1,1+k,65,2+k);
          Insert(New(PStaticText,Init(R,#3+Txt[k])))
        end;
      for k := 0 to 4 do {Вставляем кнопки:}
        begin
          R.Assign(X+k*DX,6,X+k*DX+L,8);
          Insert(New(PButton,
            Init(R,But[k],cmCan+k,bfNormal)))
        end;
      SelectNext(False); {Активизируем первую кнопку}
    end;
  Control := DeskTop^.ExecView(D); {Выполняем диалог}
end; {Control}

```

Сначала создается диалоговое окно с заданными размерами (чтобы программе стал доступен тип *TDialog*, укажите в предложении *Uses* модуль *Dialogs*). Затем в цикле

```
for k := 0 to 3 do
```

в окно вставляется поясняющий текст (см. рис.15.10). Этот текст не связан с диалогом и называется статическим. Для вставки статической строки в любой видимый элемент используется конструктор *TStaticText.Init*, которму в качестве параметров передаются координаты строки и сама строка. Как Вы уже могли заметить, идентификаторы объектов в Turbo Vision начинаются на букву *T*, а идентификаторы типов-указателей на экземпляры этих объектов начинаются на букву *P*. Таким образом, *PStaticText* — это тип-указатель на экземпляр объекта *TStaticText*, поэтому оператор

```
Insert(New(PStaticText, Init(R,'Текст'))
```

помещает строку «Текст» на место, заданное координатами переменной *R*. Отметим, что если строка начинается на символ *#3*, то при выводе на экран она будет размещаться в центре прямоугольника *R*. Мы используем это соглашение и дополняем каждую выводимую строку этим символом.

В цикле

```
for k := 0 to 4 do {Вставить кнопки:}
```

в окно вставляются пять кнопок. При их инициации используется то обстоятельство, что определенные нами команды *cmCan*, ..., *cmAdd* образуют непрерывное множество [205..209].

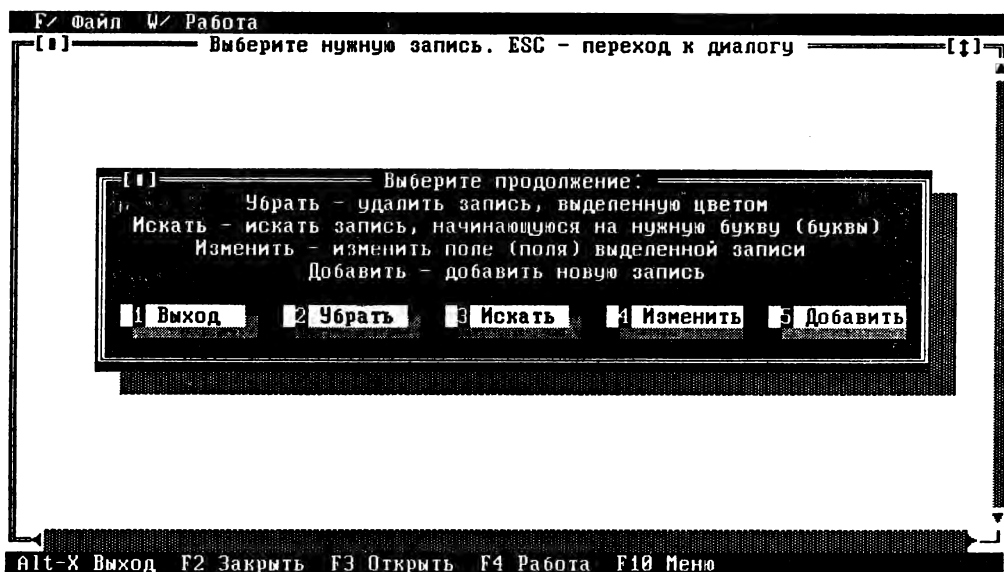


Рис.15.10. Диалоговое окно функции Control

Особо следует остановиться на операторе

```
SelectNext(False); {Активизируем 1-ю кнопку}
```

Дело в том, что по умолчанию активизируется тот элемент диалогового окна, который задан (вставлен в окно) последним. Чтобы изменить активность по умолчанию, используется вызов процедуры *SelectNext*, которая смещает активность к следующему элементу. Так как элементы образуют замкнутую цепь (от последнего элемента активность переходит к первому), параметр обращения к этой процедуре указывает направления смещения: если он имеет значение *False*, активным станет следующий в цепи элемент, если *True* — предыдущий.

Прежде, чем Вы попытаете запустить эту программу на счет, внесем в нее несколько изменений. Во-первых, пора убрать имитацию данных, показываемых в окне просмотра. Для этого в процедуре *TInterior.ReadFile* необходимо удалить строки

```
s := copy(ParamStr(0),1,pos('.',ParamStr(0))+'pas';
assign(f,s);
.....
exit;
```

Надеюсь, что Вы заблаговременно подготовили остальной текст этого метода, если это не так, вставьте операторы

```
seek (DataFile, 0);
while not (EOF (DataFile) or LowMemory) do
begin
.....
end;
```

```
Location := 0
```

(см. с.369).

Во-вторых, обратили ли Вы внимание на то, что в процедуре *TNotebook.Work* (см. с.361) указатель *PW* иницируется оператором

```
PW := New(PWorkWin, Init(R));
```

а динамическая память, выделенная для размещения экземпляра объекта *TWorkWin*, не возвращается обратно в кучу? Если да, то у Вас есть хорошие шансы избежать многих неприятностей при программировании в среде Turbo Vision. Конечно же, нам следовало где-то в программе позаботиться об удалении ненужного нам экземпляра объекта. Чтобы не усложнять программу, я не стал этого делать: если вставить оператор

```
Dispose(PW, Done)
```

сразу за оператором

```
DeskTop^.Insert(PW)
```

то вновь созданное окно будет тут же удалено с экрана, поэтому оператор *Dispose* нужно разместить в обработчике событий *TNotebook.HandleEvent* (подумайте, где именно).

После включения диалогового окна в цепочку действий, связанных с инициацией *PW*, появилась возможность приостановить исполнение программы в процедуре *Work*: вместо оператора

```
DeskTop^.Insert(PW)
```

вставьте следующие строки:

```
Control := DeskTop^.ExecView(PW);
Dispose(PW, Done)
```

и добавьте описание переменной *Control*:

```
var
.....
Control: Word;
```

В отличие от процедуры *Insert* процедура *ExecView* не только помещает видимый элемент на экран, но и приостанавливает дальнейшее исполнение программы *Work* до тех пор, пока не закончится диалог с пользователем.

И, наконец, еще одно усовершенствование. Работа с программой станет удобнее, если сразу после чтения файла с данными она перейдет к их показу. Реализовать это очень просто: добавьте вызов процедуры *Work* в процедуру *FileOpen* следующим образом:

```
Procedure TNotebook.FileOpen;
.....
begin
.....
if OpFileF then
```

```

begin
    .....
    Work          {Переходим к работе}
end;
.....
end; {FileOpen}

```

Если Вы внесете в программу все описанные изменения и запустите ее на счет¹, то при попытке выйти из режима просмотра на экране будет развернуто диалоговое окно, показанное на рис.15.10. «Нажатие» на любую кнопку этого окна не приводит ни к каким последствиям — наше окно пока откликается только на стандартную команду *cmClose*, связанную с клавишей *Esc*.

15.14. ОБРАБОТКА КОМАНД ПОЛЬЗОВАТЕЛЯ

Обработчик событий диалогового окна поддерживает только стандартные команды *cmClose*, *cmOk*, *cmCancel*. Как заставить его реагировать на команды пользователя? Вы правы: нужно перекрыть стандартный обработчик событий.

Введем новый объект *TDlgWin* как потомок объекта *TDialog* и перекроем его метод *HandleEvent*:

```

type
    PDlgWin = ^TDlgWin;
    TDlgWin = object (TDialog)
        Procedure HandleEvent(var Event: TEvent); Virtual;
    end;

```

В новом методе следует сначала вызвать стандартный обработчик, а затем проанализировать событие: если оно не очищено и содержит команду, значит была нажата какая-то командная кнопка, и нам следует заставить обработчик закрыть окно и вернуть эту команду как результат диалога с пользователем:

```

Procedure TDlgWin.HandleEvent(var Event: TEvent);
{Обработчик событий для основного диалогового окна}
begin
    Inherited HandleEvent(Event);
    if Event.What = evCommand then
        EndModal(Event.Command) {Закрыть окно и вернуть команду}
end;

```

Метод *EndModal* используется для того, чтобы завершить работу с диалоговым окном и вернуть команду в программу, использующую это окно. Измените описание переменной *D* в функции *Control* на

var

¹ Файл с данными *DataType* пока еще не существует. Чтобы программа смогла нормально работать, в диалоговом окне открытия файла укажите произвольное имя, например *MYDATA*. После завершения работы программы будет создан пустой файл *MYDATA.DAT*.


```
.....
D: PDlgWin;
```

и обращение к методу *Init*:

```
D := New(PDlgWin, Init(...));
```

и вновь запустите программу: теперь нажатие на любую кнопку диалогового окна приведет к его закрытию.

15.15. РЕДАКТИРОВАНИЕ И ДОБАВЛЕНИЕ ЗАПИСЕЙ

Для редактирования и добавления записей создадим окно, показанное на рис.15.11.

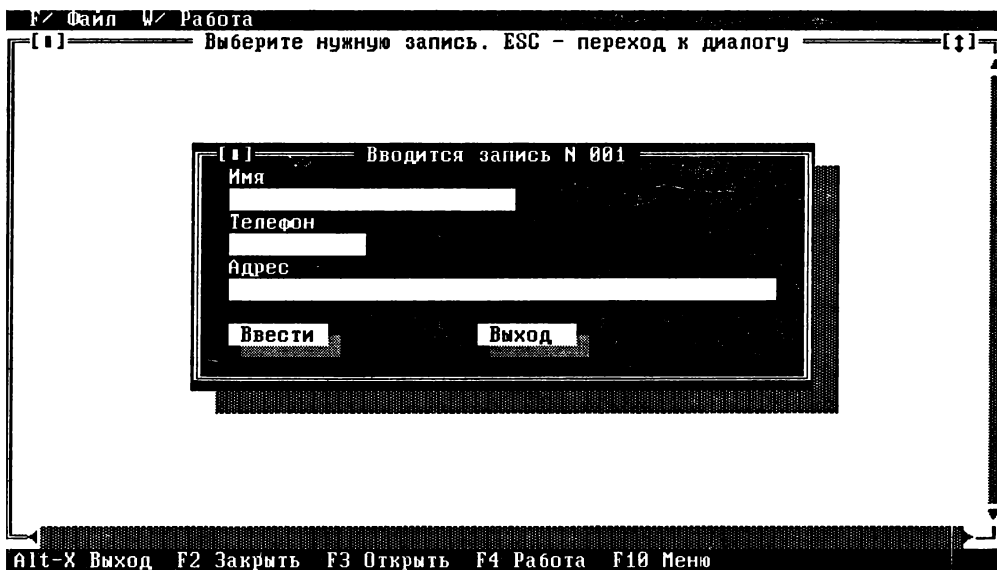


Рис.15.11. Окно ввода/редактирования записей

Режим редактирования отличается от режима добавления записей двумя обстоятельствами: во-первых, в режиме редактирования поля ввода данных окна должны содержать текст, взятый из редактируемой записи, а в режиме ввода эти поля пусты. Во-вторых, режим редактирования завершается сразу после нажатия на клавишу *Enter*, в то время как в режиме ввода нажатие на эту клавишу означает добавление к файлу текущей записи и переход к вводу следующей: режим ввода завершается командой *cmClose* (клавиша *Esc*). С учетом этого оба режима реализуются в рамках одной процедуры *AddItem (Edit)*, а параметр *Edit* указывает нужный режим: если *Edit = True*, реализуется режим редактирования, если *False* — режим добавления записей. Вот текст этой процедуры:

```
Procedure AddItem(Edit: Boolean);
{Добавляет новый или редактирует старый элемент данных}
const
  y = 1;
```

```

dy= 2;
L = LName+LPhone+LAddr;
var
  Data: DataType;
  R: TRect;
  InWin: PDialog;
  BName, BPhone, BAddr: PInputLine;
  Control: Word;
  OldCount: Word;
  s: String;
  p: PString;
begin
  Seek(DataFile, FileSize(DataFile));           {Добавляем записи
                                                    в конец файла}
  repeat {Цикл ввода записей}
    if Edit then {ГОТОВИМ заголовок}
      s := 'Редактирование:'
    else
      begin
        Str(FileSize(DataFile)+1, s);
        while Length(s) < 3 do
          s := '0'+s;
          s := 'Вводится запись N '+s
        end;
        FillChar(Data, SizeOf(Data), ' ');       {Заполняем поля
                                                    пробелами}
        R.Assign(15, 5, 65, 16);
        InWin := New(PDialog, Init(R, s));       {Создаем окно}
        with InWin^ do
          begin                                  {Формируем окно:}
            R.Assign(2, y+1, 2+LName, y+2);
            BName := New(PInputLine, Init(R, LName));
            Insert(BName);                      {Поле имени}
            R.Assign(2, y, 2+LName, y+1);
            Insert(New(PLabel,
              Init(R, 'Имя', BName)));
            R.Assign(2, y+dy+1, 2+LPhone, y+dy+2);
            BPhone := New(PInputLine,
              Init(R, LPhone));
            Insert(BPhone);                     {Поле телефона}
            R.Assign(2, y+dy, 2+LPhone, y+dy+1);
            Insert(New(PLabel,
              Init(R, 'Телефон', BPhone)));
            R.Assign(2, y+2*dy+1, 2+LAddr, y+2*dy+2);
            BAddr := New(PInputLine, Init(R, LAddr));
            Insert(BAddr);                      {Поле адреса}
            R.Assign(2, y+2*dy, 2+LAddr, y+2*dy+1);
            Insert(New(PLabel,

```

```

Init(R, 'Адрес', BAddr));
{Вставляем две командные кнопки:}
R.Assign(2, y+3*dy+1, 12, y+3*dy+3);
Insert(New(PButton,
Init(R, 'Ввести', cmOK, bfDefault)));
R.Assign(2+20, y+3*dy+1, 12+20, y+3*dy+3);
Insert(New(PButton,
Init(R, 'Выход', cmCancel, bfNormal)));
SelectNext(False) {Активизируем первую кнопку}
end; {Конец формирования окна}
if Edit then with Data do
begin {ГОТОВИМ начальный текст:}
p := PS^.At(Location); {Читаем данные из записи}
s := p^;
Name := copy(s, 1, LName);
Phone := copy(s, succ(LName), LPhone);
Addr := copy(s, succ(LName+LPhone), LAddr);
InWin^.SetData(Data) {Вставляем текст в поля ввода}
end;
Control := DeskTop^.ExecView(InWin); {Выполняем диалог}
if Control=cmOk then with Data do
begin
if Edit then
DeleteItem; {Удаляем старую запись}
Name := BName^.Data^;
Phone := BPhone^.Data^;
Addr := BAddr^.Data^;
s[0] := chr(L);
FillChar(s[1], L, ' ');
move(Name[1], s[1], Length(Name));
move(Phone[1], s[succ(LName)], Length(Phone));
move(Addr[1], s[succ(LName+LPhone)], Length(Addr));
OldCount := PS^.Count; {Прежнее количество записей}
PS^.Insert(NewStr(s)); {Добавляем в коллекцию}
{Проверяем добавление}
if OldCount <> PS^.Count then
Write(DataFile, Data) {Да - добавляем в файл}
end
until Edit or (Control=cmCancel);
Draw
end; {AddItem}

```

Вначале указатель файла смещается в самый конец, подготавливая добавление записей (судя по всему, режим добавления будет использоваться гораздо чаще, чем режим редактирования). Затем формируется заголовок окна и само окно. Операторы

```
if Edit then with Data do
```

```
begin {Готовим начальный текст;}
.....
end;
```

готовят начальное состояние полей ввода в режиме редактирования. Оператор

```
InWin^.SetData(Data)
```

помещает подготовленный текст в нужные поля. При обращении к процедуре *SetData* данные должны быть предварительно подготовлены в строгом соответствии с порядком создания диалоговых полей в окне и типом их данных. Поскольку в нашем случае формат данных в полях ввода окна совпадает с форматом файловых данных, мы можем использовать одну и ту же переменную как для работы с файлом, так и для установки начальных значений диалоговых полей.

В самом общем случае пользователь должен объявить новый тип, соответствующий формату помещаемых в окно данных, и использовать выражение этого типа в качестве параметра обращения к процедуре *SetData*. Например, если бы в нашем окне было предусмотрено только одно поле ввода «Телефон», то установку данных можно было бы осуществить таким оператором:

```
InWin^.SetData(DataType.Phone)
```

где *DataType.Phone* — выражение типа *String* [*LPhone*].

Контроль за соответствием типа устанавливаемых данных порядку объявления и типу данных диалоговых полей полностью возлагается на программиста.

В операторах

```
if Control=cmOk then with Data do
begin
.....
end
```

данные, полученные из диалогового окна, помещаются сначала в отсортированную коллекцию, а затем — в файл. С помощью оператора

```
if OldCount <> PS^.Count then
```

проверяется изменение количества данных в коллекции (напомню, что в отсортированную коллекцию можно поместить только уникальную запись). Если количество записей в коллекции изменилось, значит новая запись не совпадает ни с одной из уже имеющихся и ее следует поместить в файл.

Операторы

```
if Edit then
DeleteItem; {Удаляем старую запись}
```

предварительно удаляют старую запись с помощью обращения к процедуре *DeleteItem*.

15.16. УДАЛЕНИЕ ЗАПИСИ

При реализации режима удаления записи нам нужно учесть тот факт, что порядок следования записей в файле и коллекции может быть различным. Поэтому в процедуре *DeleteItem* организуется цикл поиска в файле удаляемой записи:

Procedure DeleteItem;

{Удаляет указанный в Location элемент данных}

var

D: Integer;

PStr: PString;

s: String;

Data: DataType;

begin

PStr := PS^.At(Location); {Получаем текущую запись}

s := copy(PStr^,1,LName);

seek(DataFile,0);

D := -1; {D - номер записи в файле}

repeat {Цикл поиска по совпадению поля Name:}

inc(D);

read(DataFile,Data);

with Data **do while** Length(Name) < LName **do**

Name := Name+' '

until Data.Name=s;

seek(DataFile,pred(FileSize(DataFile)));

read(DataFile,Data); {Читаем последнюю запись}

seek(DataFile,D);

write(DataFile,Data); {Помещаем ее на место удаляемой}

seek(DataFile,pred(FileSize(DataFile)));

truncate(DataFile); {Удаляем последнюю запись}

with PS^ **do** D := IndexOf(At(Location));

PS^.AtFree(D); {Удаляем строку из коллекции}

Draw {Обновляем окно}

end; {DeleteItem}

15.17. РЕЖИМ ПОИСКА ЗАПИСИ

Для поиска нужной записи сформируем диалоговое окно, показанное на рис.15.12.

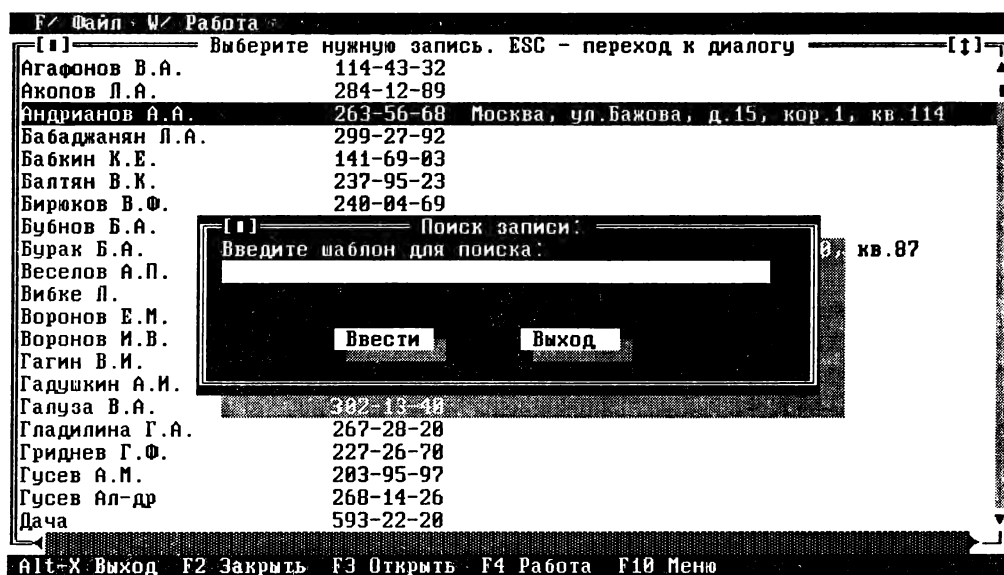


Рис.15.12. Окно ввода шаблона поиска

С помощью этого окна пользователь может задать несколько начальных букв, используемых как ключ для поиска записи. Получив данные из этого окна, процедура *SearchItem* организует поиск первой от начала коллекции строки, для которой не выполняется условие

$$\text{Pattern} \geq \text{Item}$$

где *Pattern* — образец поиска, *Item* — текущая строка коллекции. Найденная строка указывается как текущая в поле *Location* и организуется вывод соответствующего текста в окне просмотра.

В реализации процедуры *SearchItem* указанная проверка осуществляется для строк, предварительно преобразованных к прописным буквам с помощью внутренней процедуры *UpString*, т.е. поиск игнорирует возможную разницу в высоте букв шаблона и строк коллекции.

Procedure SearchItem;

{Ищет нужный элемент}

Function UpString(s: String): String;

{Преобразует строку в верхний регистр}

var

k: Integer;

begin

for k := 1 to Length(s) do

if s[k] in ['a'..'z'] then

s[k] := chr(ord('A')+ord(s[k])-ord('a'))

else if s[k] in ['a'..'н'] then

s[k] := chr(ord('A')+ord(s[k])-ord('a'))

else if s[k] in ['п'..'я'] then

s[k] := chr(ord('P')+ord(s[k])-ord('p'));

```

UpString := s
end; {UpString}
var
  InWin: PDialog;
  R: TRect;
  s: String;
  p: PInputLine;
  k: Word;
begin {SearchItem}
  R.Assign(15,8,65,16);
  InWin := New(PDialog,
  Init(R, 'Поиск записи:'));
  with InWin^ do
    begin
      R.Assign(2,2,47,3);
      p := New(PInputLine, Init(R,50));
      Insert(p);
      R.Assign(1,1,40,2);
      Insert(New(PLabel, Init(R,
        'Введите образец для поиска:',p)));
      R.Assign(10,5,20,7);
      Insert(New(PButton,
        Init(R, 'Ввести', cmOk, bfDefault)));
      R.Assign(25,5,35,7);
      Insert(New(PButton,
        Init(R, 'Выход', cmCancel, bfNormal)));
      SelectNext(False)
    end;
  if DeskTop^.ExecView(InWin) = cmCancel then
    exit;
  s := p^.Data^;
  Location := 0;
  while (UpString(s) >= UpString(PString(PS^.At(Location))^))
    and (Location < pred(PS^.Count)) do
    inc(Location);
  if (Location < Delta.Y) or
    (Location > Delta.Y+pred(Size.Y)) then
    ScrollTo(Delta.X, Location)
  else
    Draw
  end; {SearchItem}

```

15.18. ИТОГИ

Итак, мы завершили создание диалоговой программы, обслуживающей электронную «записную книжку». В ходе ее реализации Вы познакомились с некоторыми возможностями диалоговой среды Turbo Vision. Я не ставил себе целью дать здесь подробное описание всех или даже использованных

в программе средств Turbo Vision — этому посвящены остальные главы этой части книги. В частности, вне рамок примера остались такие важные механизмы, как потоки, ресурсы, препроцессорные и постпроцессорные события и многие другие возможности Turbo Vision. Однако уже рассмотренные средства свидетельствуют о том, что программирование в Turbo Vision существенно отличается от традиционных методов создания программ, и это отличие является следствием широкого использования в Turbo Vision механизмов объектно — ориентированного программирования.

Наша программа, насчитывающая всего около 600 строк, обеспечивает весьма высокий уровень диалога с пользователем: в ней используются командные клавиши, «выпадающие» меню, удобные диалоговые окна, поддержка мыши. Думаю, что вряд ли каким — либо другим способом мы смогли бы создать столь сложную диалоговую среду программой такого объема. Таким образом, Turbo Vision является превосходным инструментом для разработки диалоговых программ, ориентированных на текстовый режим работы экрана. Использование этой среды в Ваших программах резко сократит сроки их создания и повысит качество.

Разумеется, созданная программа далека от совершенства, однако даже в этом виде она, как показывает мой собственный опыт, может быть достаточно полезной. При желании ее можно взять за основу создания более сложной информационно — поисковой системы.

ОБЩАЯ ХАРАКТЕРИСТИКА ОБЪЕКТОВ

Все свойства Turbo Vision заключены в полях и методах входящих в библиотеку объектов. Каждый объект предназначен для тех или иных целей, поэтому, если Вы хотите использовать какие-то возможности Turbo Vision, Вы должны создать и использовать экземпляры объектов с нужными Вам свойствами. Этот процесс не представляет никакой сложности: вспомните обычные переменные или константы Турбо Паскаля; если, например, в программе понадобится осуществить преобразование строковых данных, Вы объявляете переменную типа *String*, для реализации математических вычислений — переменную типа *Real* и т.д. Точно также обстоит дело и с объектами: для создания окна Вы можете объявить переменную типа *TWindow*, для использования отсортированной коллекции строк — переменную типа *TStringCollection* и т.д. Например:

```
var
  MyWindow: TWindow;
  MyCollection: TStringCollection;
```

Однако для многих практических применений Вы можете не найти среди объектов Turbo Vision такого, который бы имел все нужные Вам свойства. В этом случае Вы должны создать новый объект как потомок от какого-либо стандартного объекта Turbo Vision и наделить его дополнительными свойствами. Если, например, вновь создаваемое окно предназначено для вывода отсортированных текстовых строк, Вы можете изменить его конструктор *Init*, обеспечив в нем создание и использование экземпляра объекта *TStringCollection*:

```
type
  MyWindowType = object (TWindow)
    MyCollection: PStringCollection;
    Constructor Init;
  end;
Constructor MyWindowType.Init;
.....
begin
  .....
  MyCollection := New(PStringCollection, Init(...));
  .....
end;
```

Чтобы вновь создаваемый объект был по возможности простым в реализации, он должен наследовать от объекта-родителя максимальное количество нужных ему полей и методов. Таким образом, ключом к использованию Turbo Vision является знание тех свойств, которыми обладают ее стандартные объекты. Изучению объектов посвящена фактически вся

эта часть книги, в этой главе мы дадим наиболее общую их характеристику.

16.1. СТРУКТУРА ОБЪЕКТОВ

Все множество объектов Turbo Vision строится на базе трех примитивных объектов: *TPoint*, *TRect* и *TObject*. Примитивными они называются потому, что не являются потомками каких-либо других объектов, но используются или могут использоваться как основа порождаемых от них деревьев родственных типов (рис.16.1).



Рис.16.1. Примитивные объекты Turbo Vision

Объекты *TPoint* и *TRect* используются для задания координат точки (*TPoint*) и прямоугольника (*TRect*) на экране. В Turbo Vision эти объекты не имеют потомства, но используются всеми другими объектами, связанными с созданием изображений на экране ПК. Объект *TObject* служит основой иерархии всех других объектов Turbo Vision, кроме *TPoint* и *TRect* (см. рис.16.2).

Объект *TObject* не имеет данных (полей) и содержит три метода: *Init*, *Free* и *Done*. Конструктор *Init* осуществляет распределение памяти и является основой всех других конструкторов Turbo Vision. Метод *Free* освобождает память, распределенную конструктором. Деструктор *Done* не выполняет никакой работы — это абстрактный метод, который всегда перекрывается в объектах — потомках.

16.2. АБСТРАКТНЫЕ ОБЪЕКТЫ И МЕТОДЫ

Объект *TObject* никогда не используется самостоятельно и служит основой для начинающегося с него дерева объектов. В Turbo Vision есть и другие объекты, не предназначенные для самостоятельного использования в программах. Такие объекты называются абстрактными. В правильно сконструированной программе никогда не создаются и не используются экземпляры абстрактных объектов. Эти объекты просто объединяют в себе некоторые общие для всех своих потомков поля и методы.

Во многих объектах Turbo Vision используются абстрактные методы, т.е. методы, которые не содержат исполняемых операторов и служат заголовками для аналогичных методов в объектах — потомках. Абстрактные методы всегда являются виртуальными и должны перекрываться в потомках. Например, абстрактными являются деструктор *TObject.Done*, используемый для удаления экземпляра объекта из кучи, метод *TSortedCollection.Compare*, с помощью которого осуществляется сортировка коллекций, метод *TCluster.Press*, ответственный за обработку события, связанного с выбором элемента управления в диалоговом окне. В большинстве абстрактных методов предусмотрен вызов глобальной процедуры *Abstract*, предназначенной для аварийного завершения работы программы и выдачи соответствующего диагностического сообщения.

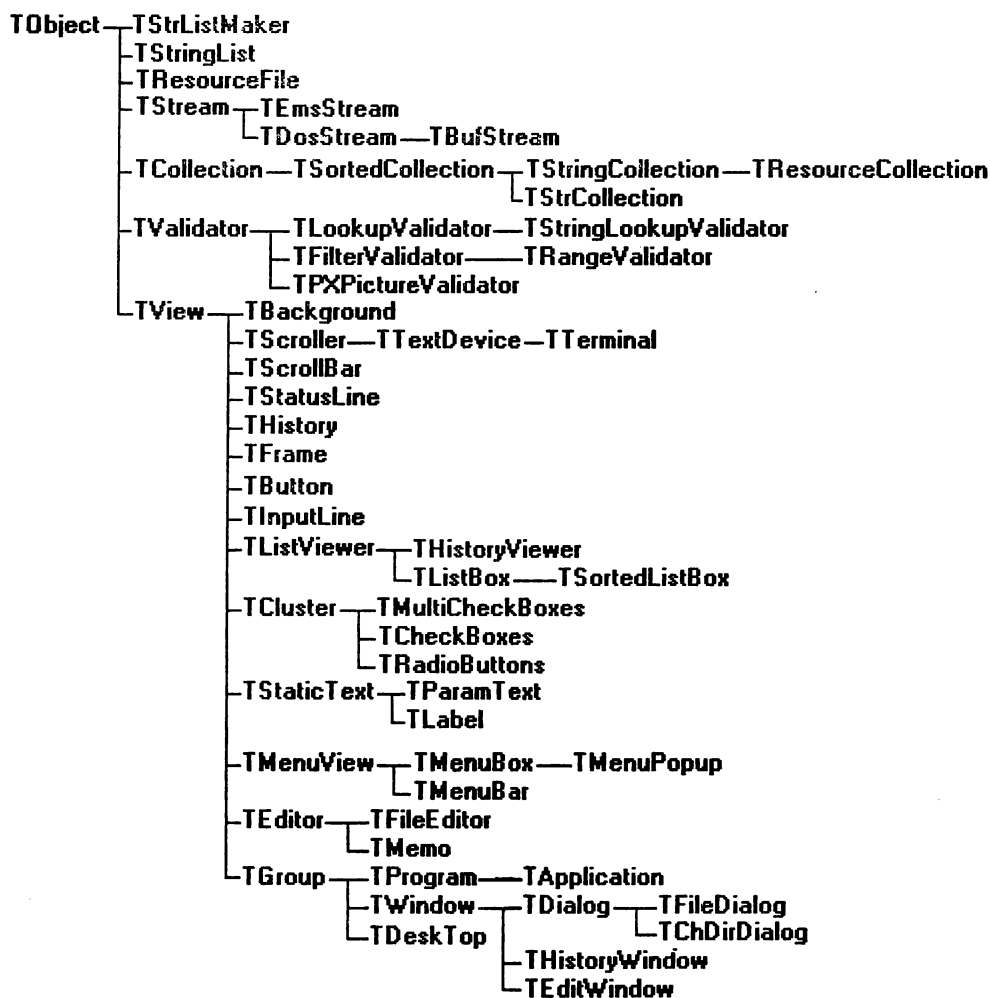


Рис.16.2. Иерархия объектов Turbo Vision

Некоторые методы не являются абстрактными, но предоставляют лишь минимум возможностей и поэтому почти всегда перекрываются в потомках. Такие методы называются псевдоабстрактными. Примером псевдоабстрактного метода может служить метод *TView.Draw*, ответственный за перерисовку видимого элемента: по умолчанию он ничего не делает и оставляет выделенную ему часть экрана без изменений.

16.3. ФУНКЦИОНАЛЬНОСТЬ ОБЪЕКТОВ

В функциональном отношении все потомки объекта *TObject* делятся на две группы: видимые элементы и невидимые объекты (рис.16.3).

Видимые элементы могут быть терминальными видимыми объектами или группами видимых элементов. Любой элемент группы в свою очередь

может быть группой элементов, однако все группы в конце концов заканчиваются терминальными видимыми объектами.

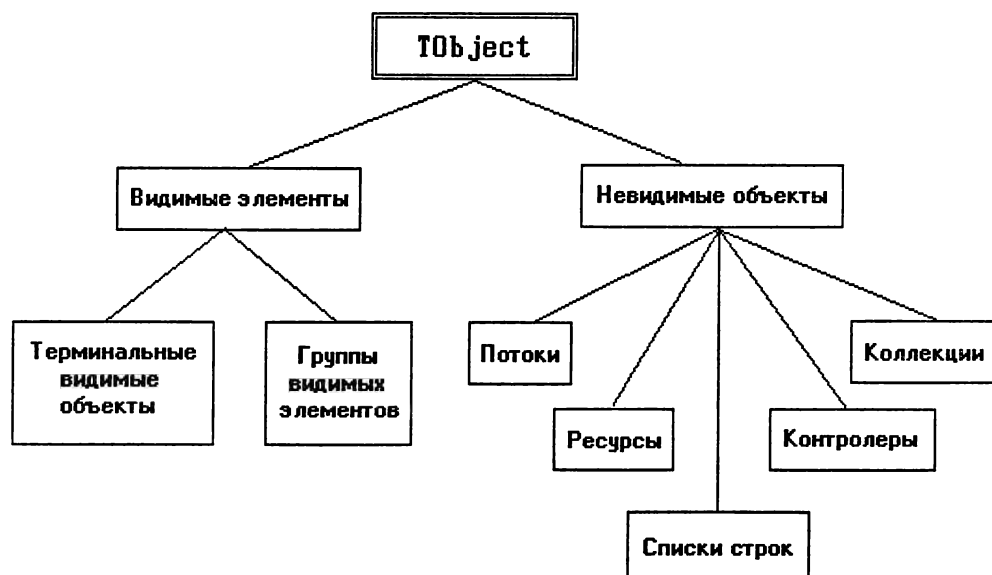


Рис. 16.3. Видимые и невидимые объекты

Все терминальные видимые объекты (или просто видимые объекты) являются потомками объекта *TView*, который придает им специальные свойства, отсутствующие у невидимых объектов. Объект *TView* — это прямой потомок *TObject* (см. рис.16.2), представляющий собой пустую прямоугольную область экрана. В программе почти никогда не используются экземпляры этого объекта, однако от него прямо или косвенно порождаются все видимые элементы — как стандартные для Turbo Vision, так и создаваемые программистом. Каждый порождаемый от *TView* видимый объект владеет прямоугольной частью экрана и наследует его псевдоабстрактные методы *Draw* и *HandleEvent*. Метод *Draw* является основой специализированных методов, обеспечивающих создание изображений. Метод *HandleEvent* — это центральный метод, с помощью которого реализуется цепочка связанных друг с другом обработчиков событий. Каждый видимый элемент может обрабатывать связанные с ним события — это является одним из важнейших отличительных свойств видимых элементов.

Семейство невидимых объектов включает в себя потоки, ресурсы, контролеры, коллекции и списки строк. Невидимые объекты по существу представляют собой обобщение таких важных механизмов Турбо Паскаля, как файлы и массивы, учитывающее особенности объектно-ориентированных программ. Вы можете использовать невидимые объекты в любых программах, в том числе и не пользующихся изобразительными возможностями Turbo Vision. В частности, они могут оказаться полезными при разработке программ, рассчитанных на графический режим работы

дисплея (видимые объекты Turbo Vision рассчитаны исключительно на текстовый режим работы).

16.4. ОБЗОР ВИДИМЫХ ЭЛЕМЕНТОВ

Видимый объект — это любой объект, который является потомком *TView* и может быть изображен на экране в текстовом режиме работы дисплея. Важным свойством видимых объектов является возможность объединения нескольких видимых объектов в группу, рассматриваемую как единое целое. Любым элементом группы в свою очередь может быть новая группа, в которую могут входить другие группы и т.д., однако любая группа видимых элементов всегда заканчивается терминальными видимыми объектами.

16.4.1. Группы видимых элементов

Абстрактная группа

Абстрактная группа — это объект *TGroup*. Этот объект, вообще говоря, не является абстрактным, однако его экземпляры практически никогда не используются в программах: основное назначение объекта — служить основой иерархии родственных объектов пользователя. Хотя группа владеет прямоугольным участком экрана, она становится видимой только за счет визуализации своих элементов. Элементы группы организуются в связанный список: каждый элемент имеет поле *Owner*, содержащее указатель на своего владельца, и поле *Next*, с помощью которого указывается следующий элемент списка.

Панель экрана

Панель экрана — это объект *TDesktop*, создающий фоновое изображение рабочей зоны экрана. *TDesktop* является прямым потомком *TGroup* и, следовательно, может владеть произвольным набором видимых элементов. Заполнение панели экрана осуществляется другими видимыми элементами, такими как *TWindow*, *TDialog* и т.п. Обычно владельцем группы *TDesktop* является программа — объект *TApplication* или его потомки.

Программы

Все программы, использующие средства Turbo Vision, должны порождаться от объекта *TProgram* или его единственного потомка *TApplication*. Оба объекта создают на экране стандартное изображение строки меню, панели экрана и строки статуса, т.е. являются владельцем группы, состоящей из *TMenuBar*, *TDesktop* и *TStatusLine* (рис.16.4).

TApplication отличается от своего родителя только методами *Init* и *Done*, с помощью которых осуществляется инициация основных подсистем Turbo Vision и их закрытие:

```
Constructor TApplication.Init;
```

```

begin
  InitMemory;
  InitVideo;
  InitEvents;
  InitSysError;
  InitHistory;
  TProgram.Init
end;
Destructor TApplication.Done;
begin
  TProgram.Done;
  DoneHistory;
  DoneSysError;
  DoneEvents;
  DoneVideo;
  DoneMemory
end;

```

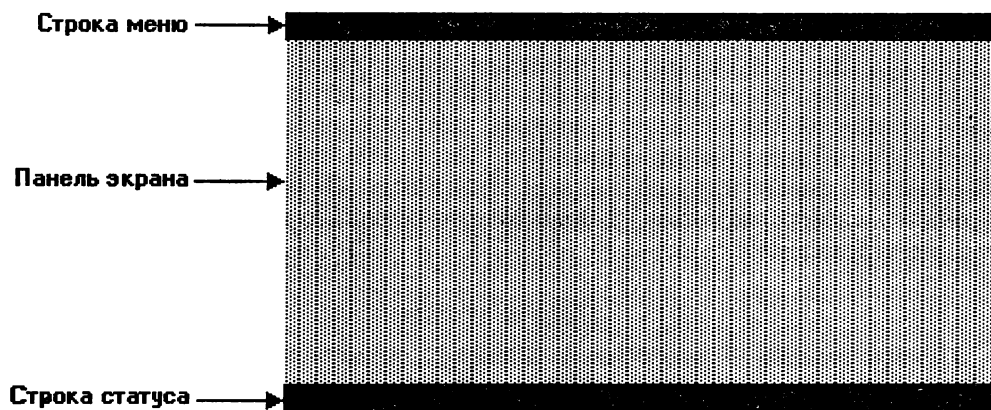


Рис.16.4. Стандартная группа TProgram

Обычно программа пользователя связывается с экземпляром *TApplication* или его потомка. Если Вас не устраивает стандартная последовательность инициации и закрытия подсистем Turbo Vision, Вы должны создать и использовать потомка от *TProgram*.

Окна

Для создания и использования окон в Turbo Vision предусмотрен объект *TWindow*. Обычно этот объект владеет объектом *TFrame* и поэтому очерчивается прямоугольной рамкой со стандартными кнопками изменения размера и закрытия. Вы можете перемещать окно, изменять его размеры и закрывать, используя методы *TView* или перекрывая их собственными методами. Если окно имеет номер от 1 до 9, его можно сделать активным (выбрать) командными клавишами *Alt-N*, где *N* — номер окна. Если окно имеет несколько видимых элементов, его обработчик событий интерпре-

тирует нажатие на клавиши *Tab* и *Shift–Tab* как команду активизации следующего (предыдущего) видимого элемента.

Диалоговые окна

Объект *TDialog* порожден от *TWindow* и предназначен для реализации взаимодействия с пользователем. В отличие от *TWindow* диалоговое окно не может изменять свой размер (но может перемещаться по экрану). Его обработчик событий генерирует команду *cmCancel* в ответ на нажатие клавиши *Esc* (или воздействие мышью на кнопку закрытия) и команду *cmDefault* в ответ на нажатие *Enter*.

16.4.2. Терминальные видимые объекты

Рамки

Для создания рамки окна и кнопок изменения размера окна и его закрытия предназначен объект *TFrame* (рис.16.5).

Этот объект никогда не используется самостоятельно, но только в группе, связанной с объектом *TWindow*. Последний имеет поле *Frame* с указателем на ассоциированный объект *TFrame*.

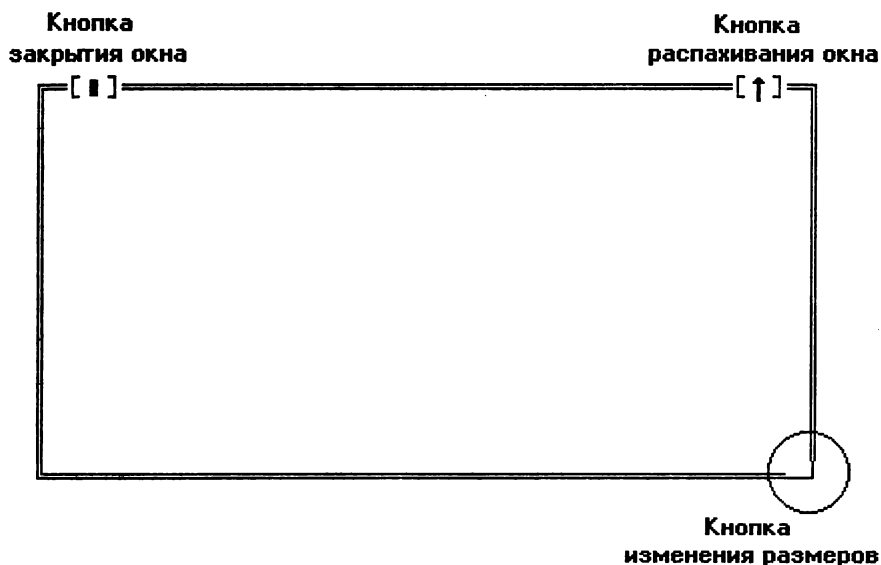


Рис.16.5. Рамка окна (объект *TFrame*)

Кнопки

Объект *TButton* — это прямоугольник с надписью, имитирующий кнопку панели управления. Обычно *TButton* является элементом группы *TDialog* и «нажатие» на кнопку инициирует событие, связанное с какой-либо стандартной командой или командой пользователя.

Кластеры

Кластеры представляют собой прямоугольные видимые элементы, имитирующие несколько зависимых или независимых кнопок (см. рис.16.6).

Устройство вывода:

☐ Экран
☒ Принтер
☐ Файл

а)

Вид копии:

☒ Инверсная
☒ Горизонтальная
☐ Перевернутая

б)

Рис.16.6. Примеры кластеров: а) с зависимыми кнопками; б) с независимыми кнопками

Для создания и использования кластера предусмотрен абстрактный объект *TCluster* и его потомки *TRadioButtons*, *TCheckBoxes* и *TMultiCheckBoxes*. Объект *TCluster* имеет 16-разрядное поле *Value*, каждый разряд которого связан со своей кнопкой в кластере. Его потомки отличаются способом изменения состояния этих разрядов: *TRadioButtons* устанавливает содержимое разряда выбранной кнопки в 1, предварительно очистив поле *Value*, а *TCheckBoxes* переключает его содержимое, сохраняя остальные разряды неизменными. Многопозиционная кнопка *TMultiCheckBoxes* может иметь от одного до 8 связанных с ней разрядов в поле *Value*. Количество разрядов зависит от количества возможных состояний кнопки, которых может быть до 256.

Таким образом, кластер может содержать до 16 кнопок и выбирать один из возможных вариантов (объект *TRadioButtons*) или несколько вариантов (*TCheckBoxes* и *TMultiCheckBoxes*). Для переключения кнопок используется мышь или клавиатура. В последнем случае клавишей *Tab* выбирается нужный кластер, а клавишами смещения курсора — нужная кнопка; состояние выбранной кнопки меняется нажатием на клавишу *Пробел*.

Меню

Для выбора одного из нескольких возможных продолжений в диалоговых программах широко используются меню. Объект *TMenuView* и его потомки *TMenuBar* и *TMenuBox* существенно упрощают создание и использование меню в среде Turbo Vision (рис.16.7).

Любая программа в Turbo Vision всегда владеет строкой меню. Переключая *TApplication.InitMenuBar*, Вы можете использовать методы *NewSubMenu*, *NewItem* и *NewLine*, чтобы создать и использовать свою систему вложенных меню.

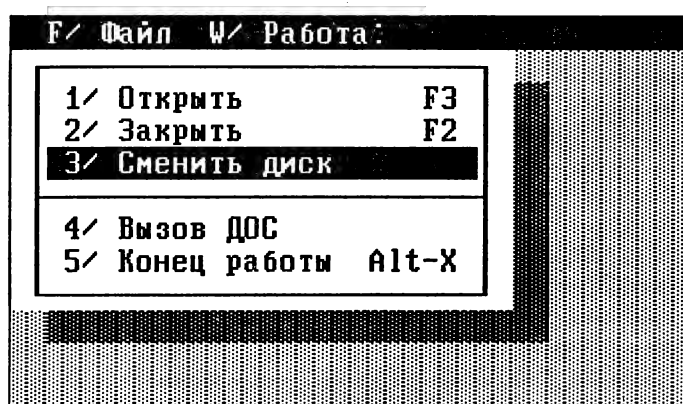


Рис.16.7. Пример меню

Строки ввода

Для ввода различных текстовых строк с клавиатуры ПК в Turbo Vision используется тип *TInputLine* (рис.16.8). Использование этого объекта дает в распоряжение пользователя мощные возможности встроенного редактора, обрабатывающего алфавитно-цифровые клавиши, клавиши перемещения курсора влево/вправо, а также клавиши *Backspace*, *Del*, *Insert*, *Home*, *End*.

Введите имя файла:

***.dat**

Рис.16.8. Пример строки ввода

Протокол ввода

Абстрактный объект *THistory* реализует механизм выбора строки ввода из списка ранее введенных строк (из протокола ввода). Объект *THistory* обычно ассоциируется с объектом *TInputLine* и становится доступным с помощью клавиши смещения курсора вниз (см. рис.16.8). Протоколы ввода запоминаются в динамической памяти в виде связанных списков и имеют идентификатор (поле *HistoryID*), позволяющий использовать один и тот же протокол в нескольких объектах *TInputLine*.

Скользящие окна (скроллеры)

С помощью объекта *TScroller* реализуется так называемое скользящее окно, т.е. окно с текстом, положение которого (текста) относительно границ окна может меняться. Условимся для краткости такие объекты в дальнейшем называть скроллерами, а процесс смещения текста в окне — скроллингом. В общем случае размеры окна скроллера могут быть значительно меньше полных размеров просматриваемого текста: средства скроллера дают возможность вывести на экран любой фрагмент текста. Обычно скроллеры объединяются в одну группу с объектами *TScrollBar*, представляющими собой горизонтальную или вертикальную полосу. Эти

полосы (полосы скроллинга) указывают положение окна относительно полных границ текста и используются для реализации скроллинга с помощью мыши. Скроллинг текста осуществляется также в ответ на нажатие клавиш смещения курсора и клавиш *PgUp*, *PgDn*, *Home*, *End*, *Ctrl-PgUp*, *Ctrl-PgDn*.

Потомки *TScroller* — объекты *TTextDevice* и *TTerminal* облегчают реализацию скользящих окон специального вида: объекты содержат методы, обеспечивающие запись текстовых строк в скроллер и чтение этих строк из него. Объект *TTerminal* отличается от своего родителя наличием внутреннего кольцевого буфера, предназначенного для запоминания текстовых строк, и соответствующих методов доступа к буферу. Размер буфера не может быть больше 64 Кбайт. Типичное применение этих объектов — реализация так называемых драйверов текстовых устройств. Например, можно связать существующий текстовый файл с объектом *TTerminal* и обеспечить непосредственное считывание строк файла в окно скроллера.

Просмотр списков

Абстрактный объект *TListViewer* предоставляет в Ваше распоряжение средства просмотра списка строк и выбора из этого списка нужной строки. Списки строк выводятся в окне, управляемом полосами скроллинга. Типичное применение объектов *TListViewer* — просмотр списка файлов. Обычно для этих целей используется потомок *TListViewer* — объект *TListBox*, с помощью которого списки файлов выводятся в одну или несколько колонок.

Статический текст

Объекты *TStaticText* — это терминальные видимые объекты, используемые для вывода текстовых сообщений. В отличие от строк, созданных непосредственным обращением к процедуре *WriteLn*, текстовая строка объекта *TStaticText* может входить в группу видимых элементов (например, окно) и управляться этой группой. Статический текст игнорирует любые события, посланные к нему. Типичное применение *TStaticText* — создание различного рода информационных окон или поясняющего текста в диалоговых окнах.

Строки статуса

Самая нижняя строка экрана в Turbo Vision представляет собой строку статуса. Эта строка создается с помощью объекта *TStatusLine*. Строка статуса обычно содержит список наиболее важных командных клавиш. Элементы строки можно также выбирать мышью. Содержание строки статуса может изменяться в зависимости от контекста программы. Для этих целей в объекте предусмотрено поле *Defs*, содержащее указатель на связанный список типа *TStatusDef*. В каждом элементе этого списка есть два поля *Min* и *Max*, задающие диапазон контекста для данного варианта строки. Контекст программы определяется полем *TView.HelpCtx*.

16.5. НЕВИДИМЫЕ ЭЛЕМЕНТЫ

16.5.1. Потоки

Поток — это обобщение процедур ввода/вывода данных. Потоки имеют все необходимые методы, позволяющие им правильно обрабатывать любые наборы данных, в том числе — экземпляры объектов Turbo Vision. Базовый абстрактный объект *TStream* служит основой для специализированных потоков. Он имеет поле *Status*, определяющее режим доступа к данным (только чтение, только запись или и чтение и запись одновременно). В объекте предусмотрено 7 абстрактных методов: *Flush*, *GetPos*, *GetSize*, *Read*, *Seek*, *Truncate* и *Write*. Это методы должны перекрываться в потомках *TStream* для того, чтобы придать потокам необходимые свойства. Перед использованием любого вновь созданного объекта в потоке его необходимо зарегистрировать. В ходе регистрации объекта ему присписывается уникальный идентификатор, позволяющий Turbo Vision автоматически распознавать тип данных, циркулирующих в потоке. Каждому стандартному объекту Turbo Vision уже присвоен уникальный номер в диапазоне от 0 до 99. Для регистрации вновь создаваемых объектов программист может использовать номера от 100 до 65535.

Потоки DOS

TDOSStream — это специализированный поток, реализующий небуферизованный ввод/вывод. Метод *Init* этого объекта позволяет создать новый или открыть уже существующий дисковый файл, задав его имя и режим доступа. В объекте перекрываются все абстрактные методы *TStream* за исключением *TStream.Flush*. В большинстве программ удобнее использовать буферизованный поток *TBufStream*, порожденный от *TDOSStream*.

Буферизованные потоки

TBufStream реализует буферизованную версию *TDOSStream*. Наличие внутреннего буфера существенно увеличивает скорость доступа к данным. Объект перекрывает абстрактный метод *TStream.Flush*, используемый для выталкивания данных из буфера. Выталкивание данных означает чтение (запись) данных из буфера перед закрытием потока и очисткой буфера.

Потоки EMS

Объект *TEMSSStream* реализует обмен данными с так называемой EMS — памятью (EMS — от *Expanded Memory Specification* — спецификация расширенной памяти, т.е. оперативная память, превышающая основные для ДОС 640 Кбайт). Новые поля этого объекта содержат обработчик EMS, число страниц, размер потока и текущую позицию внутри него.

Ресурсы

Файл ресурсов (объект *TResourceFile*) — это специальный вид потока, элементы которого могут индексироваться с помощью строковых ключей.

При сохранении в потоке очередного элемента (объекта) ему приписывается текстовая строка, которая идентифицирует этот элемент. Для быстрого поиска нужного элемента поток сохраняет в дисковом файле строки — индексы в виде отсортированной коллекции строк вместе с указателем на положение в файле индексируемого элемента и его размером.

Основное назначение файла ресурсов — обеспечение относительно простого способа доступа к диалоговым элементам. Например, вместо того, чтобы создавать диалоговое окно в точке ветвления программы, можно прочитать это окно из заранее подготовленного файла ресурсов.

16.5.2. Коллекции

TCollection реализует набор элементов, подобный массивам языка Турбо Паскаль. В отличие от массивов, содержащих элементы одного какого-либо типа, коллекции обладают свойством полиморфизма, т.е. могут хранить данные *разного* типа, в том числе и разные объекты. Коллекции размещаются в динамической памяти, поэтому их размер может быть больше 64 Кбайт. Кроме того, размер коллекции может динамически изменяться в ходе работы программы. К каждому элементу коллекции можно обратиться по его номеру (индексу).

Отсортированные коллекции

С помощью объектов *TSortedCollection* организуются коллекции, элементы которых упорядочены (отсортированы) по какому-либо признаку. Объект содержит абстрактный метод *Compare*, с помощью которого осуществляется упорядочение. Этот метод должен перекрываться для задания нужного способа сортировки элементов коллекции. Метод *Insert* обеспечивает вставку новых элементов в коллекцию с учетом принятого упорядочения. Метод *Search* осуществляет поиск заданного элемента методом двоичного поиска.

Коллекции строк

TStringCollection представляет собой модификацию своего родителя *TSortedCollection*, обеспечивая лексикографическое упорядочение строк. Метод *FreeItem* удаляет нужную строку, сохраняя упорядочение строк. Для записи новых строк в коллекцию и чтения строк из нее объект имеет методы *PutItem* и *GetItem*.

Коллекции ресурсов

Объект *TResourceCollection* порожден от *TStringCollection* и используется в объекте *TResourceFile* (файл ресурсов) для реализации коллекции ресурсов. При использовании коллекции ресурсов создается и поддерживается индексная служба в виде отсортированных строк, т.е. метод *Compare* этого объекта поддерживает лексикографическое упорядочение строк.

16.5.3. Списки строк

Объект *TStringList* реализует специальный вид строкового ресурса, в котором к строкам можно обращаться с помощью числового индекса. Поле *Count* содержит число строк в объекте. Этот объект упрощает создание многоязычных диалоговых программ, т.к. с его помощью можно обращаться к любой текстовой строке по ее индексу. В объекте предусмотрен метод *Get*, осуществляющий чтение нужной строки. Для создания списка строк и добавления к нему новых строк используется объект *TStrListMaker*.

16.5.4. Контролеры

Абстрактный объект *TValidator* и его потомки образуют группу объектов — контролеров. Общее назначение этих объектов — осуществление контроля за клавиатурным вводом пользователя. Они связываются с объектами типа *TEditor* и активизируются при выполнении *TEditor.Valid*. Получив управление, контролеры проверяют соответствие введенных данных определенным требованиям и блокируют завершение ввода, если обнаружена ошибка. Объекты — контролеры избавляют программиста от рутинной работы, связанной с программированием контроля наиболее популярных форматов вводимых данных.

TValidator

Абстрактный объект *TValidator* инкапсулирует основные свойства всех объектов — контролеров. Его метод *Valid* вызывается из одноименного метода редактора *TEditor* и обращается к абстрактному методу *TValidator.IsValid*, чтобы осуществить необходимый контроль.

TPXPictureValidator

Объект *TPXPictureValidator* проверяет введенную пользователем строку на соответствие некоторому эталонному образцу ввода. Образец задается в виде текстовой строки, подготовленной в соответствии со спецификацией СУБД *Paradox* корпорации *Borland*.

TFilterValidator

Этот объект проверяет все символы, введенные пользователем, на соответствие набору допустимых символов. Если хотя бы один введенный символ не соответствует ни одному из символов эталонного набора, метод *TFilterValidator.IsValid* вернет значение *False* и таким образом будет блокирована попытка пользователя закрыть объект *TEditor*.

TRangeValidator

Объект *TRangeValidator* порожден от *TFilterValidator*. Он преобразует символьный ввод в целое число и проверяет, находится ли оно в заданном диапазоне *Min...Max* значений. Его метод *IsValid* возвращает *True* только в том случае, когда введенные пользователем символы соответствуют

внешнему представлению целых чисел, а само введенное число N удовлетворяет условию $Max \geq N \geq Min$.

TLookupValidator

Абстрактный объект *TLookupValidator* предназначен для контроля соответствия введенной пользователем строки набору эталонных строк. Фактически он модифицирует поведение *TValidator* в случае, когда проверяется произвольная текстовая строка. В его потомках должен перекрываться метод *Lookup*, осуществляющий нужный контроль.

TStringLookupValidator

Объект *TStringLookupValidator* порожден от *TLookupValidator* и осуществляет контроль введенной пользователем строки, сравнивая ее с набором допустимых строк из отсортированной коллекции строк.

ВИДИМЫЕ ЭЛЕМЕНТЫ

Любой видимый элемент Turbo Vision имеет два важнейших свойства: он полностью контролирует изображение в пределах выделенного ему участка экрана и знает, как обрабатывать связанные с этим участком события — нажатие на клавиши или отметку мышью. Эти свойства определяются двумя псевдоабстрактными методами объекта *TView* (этот объект является родителем всех остальных видимых элементов Turbo Vision): *Draw* и *HandleEvent*. Метод *Draw* знает, как рисовать объект, но не знает, когда это нужно делать. Метод *HandleEvent*, наоборот, знает когда, но не знает как. Эти методы в наибольшей степени воплощают основной принцип программ, управляемых событиями: процесс создания изображений и процесс обработки событий — это два разных процесса в том смысле, что в первом случае мы сообщаем программе как создается изображение, а во втором — когда это нужно делать. Обработке событий посвящена следующая глава.

В этой главе мы рассмотрим некоторые детали технической реализации видимых элементов, которые дадут нам возможность разобраться в том, что именно делает видимый элемент и как он это делает. Эти сведения помогут Вам правильно использовать видимые элементы в Вашей программе.

17.1. ТЕРРИТОРИАЛЬНОСТЬ

Видимый элемент владеет прямоугольным участком экрана. Правильно сконструированный элемент обязан заполнять нужным изображением всю выделенную ему область, иначе на экране останется «мусор». Чтобы элемент мог заполнить область, он должен знать координаты закрепленного за ним участка. Эти координаты хранятся в двух полях — *Origin* и *Size*, каждое из которых имеет тип *TPoint*. Поле *Origin* задает координаты левого верхнего угла области, выделенной элементу, поле *Size* — размер этой области, т.е. показывает на каком расстоянии от *Origin* находится его правый нижний угол. Минимальный по размеру видимый элемент, который может хоть что-то вывести на экран, имеет $Size.X = Size.Y = 1$.

Объект *TPoint* крайне прост, он только определяет координаты некоторой точки на экране и не имеет никаких методов:

```
type
  TPoint = object
    X: Integer; {Горизонтальная координата}
    Y: Integer; {Вертикальная координата}
  end;
```

Координаты в Turbo Vision имеют две особенности. Во-первых, они указывают позицию на экране, начиная с 0, а не с 1, как это принято в

стандартных для Турбо Паскаля подпрограммах работы с текстовым экраном (например, *GotoXY* из модуля *CRT*). Во-вторых, все координаты задаются относительно границ той группы видимых элементов, в которой создается и используется новый элемент. В Turbo Vision любой видимый элемент входит в состав какой-то группы, поскольку все элементы в конечном счете принадлежат программе, которая сама по себе является группой.

Для указания всех четырех координат видимого элемента и действий над ними используется тип *TRect* следующего вида:

type

TRect = object

A: TPoint; {Левый верхний угол }

B: TPoint; {Правый нижний угол }

Procedure Assign(XA, YA, XB, YB: Integer);

{Назначает значения параметров полям A и B}

Procedure Copy(R: TRect);

{Устанавливает все поля, равными прямоугольнику R}

Procedure Move(ADX, ADY: Integer);

{Смещает прямоугольник, добавляя ADX к A.X, B.X и ADY к A.Y, B.Y}

Procedure Grow(ADX, ADY: Integer);

{Изменяет размер, вычитая ADX из A.X и прибавляя ADX к B.X; вычитая ADY из A.Y и прибавляя ADY к B.Y}

Procedure Intersect(R: TRect);

{Изменяет положение и размер прямоугольника до области, определенной пересечением R и текущего положения элемента}

Procedure Union(R: TRect);

{Изменяет прямоугольник до области, определяемой объединением R и текущего положения элемента}

Function Contains(P: TPoint): Boolean;

{Возвращает TRUE, если точка принадлежит элементу}

Function Equals(R: TRect): Boolean;

{Возвращает True, если положение и размеры прямоугольника R и элемента одинаковы}

Function Empty: Boolean;

{Возвращает TRUE, если элемент пустой, т.е. если его поля Size.X и Size.Y равны нулю}

end;

С помощью полей *Owner^.Origin* и *Owner^.Size* видимый элемент может определить положение и размер своего владельца, т.е. той группы, в состав которой он входит, а с помощью метода

Procedure GetExtend(var R: TRect)

получить в R свои текущие координаты (напомню, что положение и размеры большинства видимых элементов могут меняться в ходе работы программы). Обычно обращение к *GetExtend* используется перед тем, как

задать максимально возможные координаты вновь создаваемого видимого элемента. Например, если внутри окна *TWindow* нужно поместить скроллер так, чтобы он занял всю внутреннюю часть окна, можно использовать такой фрагмент:

type

```
MyWindow = object (TWindow)
.....
  Constructor Init;
.....
end;
Constructor MyWindow.Init;
var
  S: PScroller           {Указатель на скроллер}
  R: TRect;
  HS, VS: PScrollBar;    {Указатели на полосы скроллера}
.....
begin
  .....
  GetExtent(R);          {Получаем координаты окна}
  R.Grow(-1,-1);         {Уменьшаем их на 1}
  S := New(PScroller,
            Init(S, HS, VS)); {Создаем скроллер}
  Insert(S);             {Помещаем скроллер в окно}
  .....
end;
```

17.2. ВЫВОД ИЗОБРАЖЕНИЯ

17.2.1. Заполнение области

Видимый элемент может быть частично или полностью перекрыт другими видимыми элементами. Turbo Vision позволяет располагать скна на экране в произвольном порядке, в том числе и накладывая их друг на друга. С помощью метода

Procedure GetClipRect(**var** R: TRect)

видимый элемент может получить координаты минимальной площади, которую он должен заполнить в данный момент. Обращение к *GetClipRect* обычно используется в методе *Draw* и позволяет до минимума сократить время обновления информации на экране.

Следует помнить, что в правильно построенной программе никакой другой видимый элемент не должен вторгаться в область владения данного элемента. Поэтому, если видимый элемент не заполнит всю выделенную ему область, этого за него не сделает никто другой, и незаполненная область окажется «замусоренной» предыдущим выводом в эту часть экрана.

Для вывода на экран не рекомендуется использовать стандартную процедуру *Write* (*WriteLn*), т.к. она заполняет только ту часть экрана, которая необходима для вывода, в то время как длина строки в видимом элементе может быть больше строки вывода. С другой стороны, эта процедура игнорирует границы видимого элемента и может «залезть» в чужую область.

Вывод в *Turbo Vision* основан на применении методов *MoveChar*, *MoveStr* и *WriteLine*. Все три метода используют переменную типа *TDrawBuffer* в качестве буфера видеопамати. Метод *MoveChar* заполняет буфер нужным символом, например, пробелом или символом *Char* (#176) — этим символом заполняется фон панели экрана. Метод *MoveStr* переносит в буфер строку (подстроку), а метод *WriteLine* переносит буфер в видеопамать и таким образом осуществляет собственно вывод на экран.

Тип *TDrawBuffer* представляет собой массив слов:

type

```
TDrawBuffer = array [0..MaxViewWidth-1] of Word;
```

Константа *MaxViewWidth* определена в интерфейсной части модуля *Views* и устанавливает максимально возможную длину вывода (132 символа). Элементы массива задают двухбайтные последовательности, используемые в видеопамати ПК для размещения кода выводимого символа (младший байт) и его атрибутов (старший байт). Байт атрибутов определяет цвет выводимого символа и цвет фона, а также содержит признак мерцания (рис.17.1).

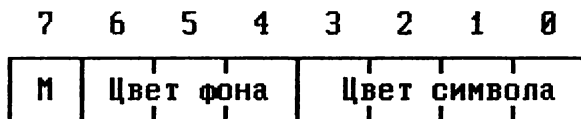


Рис.17.1. Байт атрибутов видеопамати

При обращении к методам *MoveChar* и *MoveStr* байт атрибутов задается в качестве одного из параметров обращения. Его можно получить с помощью функции *GetColor*, параметр обращения к которой определяет нужный номер элемента палитры.

17.2.2. Цвет изображения

Все цвета в *Turbo Vision* определяются системой палитр: за каждым стандартным видимым элементом закреплен свой набор условных номеров цветов, называемый палитрой. Размер палитры (количество определенных для видимого элемента цветов) зависит от функциональности элемента: чем сложнее элемент, чем больше функций он выполняет, тем богаче его палитра (каждому элементу палитры приписывается некоторая функциональность: один элемент ответственен за фон изображения, другой — за текст, третий выделяет специальные символы и т.д.). Например, для скроллера палитра состоит всего из двух элементов: первый элемент определяет цвет основного текста, второй — цвет выделенного текста. Обычно скроллер входит в качестве терминального видимого объекта в группу, палитра которой будет больше. Например, часто скроллер по-

мещается в окно *TWindow*, палитра которого насчитывает уже 8 элементов (см. рис.17.2).

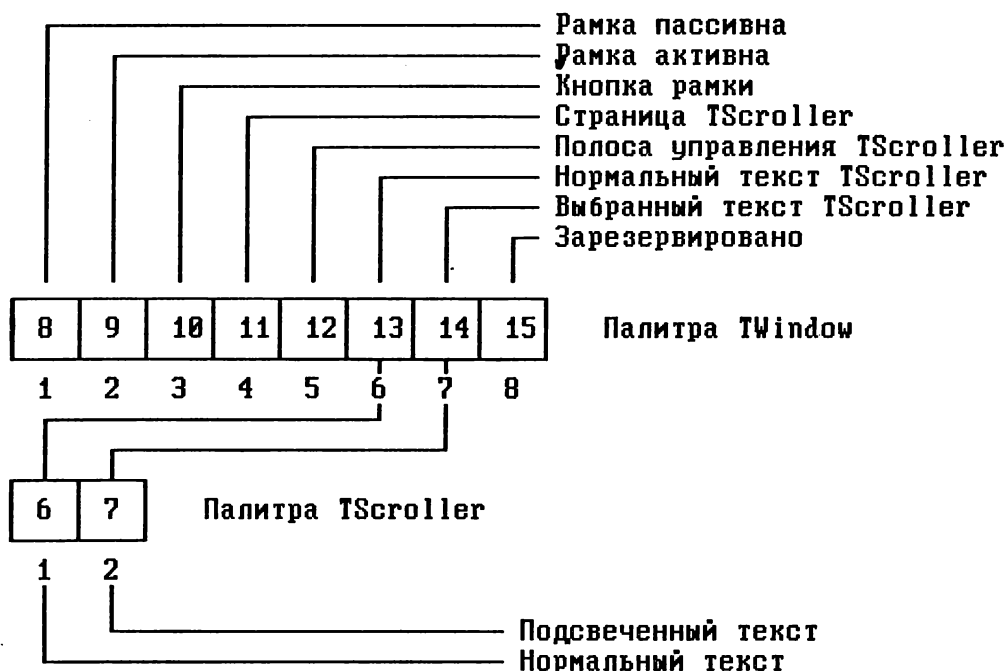


Рис.17.2. Связь палитры TScroller с палитрой TWindow

Палитры образуют систему связанных друг с другом ссылок: каждый элемент палитры содержит не какой-то конкретный цвет или его код, а целое число, указывающее на номер элемента в палитре своего владельца. Если владелец входит в группу, содержимое его палитры устанавливает связь с нужными элементами палитры этой группы и т.д. Ссылки завершаются на «владельце всех владельцев», т.е. на программе: только палитра *TProgram* и его потомков содержит не ссылки, а сами байты цветовых атрибутов.

Пусть, например, при формировании изображения в методе *Eraw* скроллера выбран первый элемент палитры (нормальный текст). Этот элемент содержит число 6, указывающее номер шестого элемента палитры владельца *TScroller*. Если владельцем скроллера является объект *TWindow*, это число означает ссылку на шестой элемент палитры *TWindow*, который содержит число 13 как ссылку на тринадцатый элемент владельца окна (рис.17.2). Если, наконец, владельцем окна является программа, то число 13 — это ссылка на тринадцатый элемент палитры *TProgram*, который содержит байт атрибутов \$1E, т.е. символы будут выводиться желтым цветом на синем фоне (рис.17.3).

Чтобы получить цвет элемента, используется обращение к функции *GetColor*. Эта функция просматривает всю цепочку ссылок от текущего видимого элемента до программы и найденный таким образом байт атрибутов из палитры *TProgram* возвращает в качестве результата. Параметром обращения к функции является номер элемента палитры видимого

В этом фрагменте в конструкторе *TMyProgram.Init* осуществляется замена 13-го элемента палитры: этот элемент отвечает за цвет основного текста скроллера (см. рис.17.3). После такого изменения во всех скроллерах программы основной текст будет выводиться белыми символами на черном фоне.

Для изменения палитры видимого элемента только одного типа нужно перекрыть его метод *GetPalette*. Допустим нам необходимо, чтобы скроллер рисовал основной текст таким же цветом, как полосы скроллера. В этом случае мы должны посмотреть, каким по счету элементом в палитре окна — владельца скроллера определяется цвет полос: в нашем примере это элемент с номером 5. Таким образом, палитра скроллера должна содержать значения 5 и 7 вместо прежних 6 и 7 (см. рис.17.2). Создадим новый объект:

```

type
  TMyScroller = object (TScroller)
    Function GetPalette: PPalette; Virtual;
  end;
Function TMyScroller.GetPalette: PPalette;
const
  NewColors = #5#7;
  NewPalette: String [2] = NewColors;
begin
  GetPalette := @NewPalette
end;

```

Существует и менее универсальный, но более простой путь изменения цвета только в одном видимом элементе. Как мы знаем, изображение этого элемента в конечном счете формируется в его методе *Draw*; если этот метод перекрывается в Вашей программе, а в некоторых случаях, например в скроллере, он должен обязательно перекрывается, то можно воздействовать на цвет непосредственно при обращениях к процедурам *MoveChar* и *MoveStr*. Например:

```

type
  MyScroller = object (TScroller)
    Procedure Draw; Virtual;
  end;
Procedure MyScroller.Draw;
var
  Color: Byte;
  ....
begin
  (* Color := GetColor(1); {Стандартный цвет скроллера} *)
  Color := $0F; {Задаем белые символы на черном фоне}
  ....
  MoveChar(.....,Color,...);
  MoveStr(.....,Color);
  ....

```

end;

В этом примере в строке комментария (* *) указывается обычный способ получения стандартного цвета основного текста скролла. Вместо этого желаемый цвет задается нужными атрибутами в переменной *Color*, которая затем используется при обращениях к процедурам *MoveChar* и *MoveStr*.

Палитра *TProgram* насчитывает 63 элемента и учитывает все возможные функциональные действия, осуществляемые видимыми элементами (см. прил.П6). Более того, этот объект на самом деле имеет три 63-элементных палитры: *CColor* (цветная палитра), *CBlackWhite* (черно-белая) и *CMonoChrome* (монохромная). В установочной секции модуля *Views* на основе тестирования аппаратных средств ПК из этих палитр выбирается рабочая палитра, которая затем и будет использоваться при формировании изображений. При необходимости Вы можете переустановить палитру *TProgram* с помощью глобальной процедуры *SetVideoMode*, например:

```
Program MyProgram;
Uses Views,.....;
var
    Main: TApplication;
    .....
begin                                {Начало основной программы}
    SetVideoMode(smBW80); {Выбрать черно-белую палитру}
    Main.Init;              {Инициация программы}
    .....
end;
```

Обращение к *SetVideoMode* должно предшествовать инициации основной программы, работающей в среде Turbo Vision. Параметром обращения к этой процедуре может быть одна из следующих констант:

```
const
    smBW80 = $002; {Черно-белый режим работы цветного адаптера}
    smCO80 = $003; {Цветной режим работы}
    smMono = $007; {Монохроматический адаптер}
```

Эти константы можно дополнять константой

```
const
    smFont8x8 = $100; {Задаёт 43/50 строк для экрана EGA/VGA}
```

для задания режима вывода 43 или 50 строк на экране дисплея, оснащённого адаптером EGA или VGA. Например:

```
SetVideoMode(smCO80+smFont8x8);
```

17.3. ГРУППЫ

Замечательным свойством видимых элементов Turbo Vision является их способность образовывать группы. Все группы являются потомками аб-

страктного объекта *TGroup*. Любая программа в конечном счете наследуется от *TProgram* или *TApplication* и, следовательно, является потомком *TGroup*, т.е. представляет собой группу.

Группа — это просто пустое окно. Главная особенность группы заключается в том, что она может управлять входящими в нее элементами. Как и любой другой видимый элемент, группа должна уметь заполнять нужным изображением выделенную ей часть экрана и обрабатывать все связанные с этой областью события. Однако в большинстве случаев группа обращается за выполнением требуемых действий к своим элементам. Например, визуализация группы происходит исключительно с помощью методов *Draw* тех элементов, которые образуют группу. С другой стороны, некоторые команды могут относиться к группе в целом. Например, группе может адресоваться команда *cmClose* (закрыть), в результате которой будет уничтожено изображение группы, т.е. очищен выделенный ей участок экрана. Для реализации этой команды группа будет последовательно передавать ее всем входящим в нее элементам.

Важно помнить, что группа обладает способностью включать в себя видимые подэлементы динамически, в ходе исполнения программы. Как правило, состав группы определяется действиями пользователя: если, например, он нажал командную клавишу, связанную с раскрытием опции главного меню, группа *TDesktop*, ответственная за рабочую зону экрана, обогащается дополнительными видимыми подэлементами «выпавшего» меню. После нажатия клавиши *Esc* эти элементы будут удалены из состава группы.

17.3.1. Создание группы и изменение ее состава

Создание группы осуществляется за счет создания экземпляра объекта — наследника *TGroup* и присоединения к нему всех видимых элементов группы. Любой видимый объект, т.е. наследник *TView*, имеет два поля: *Owner* и *Next*. Поле *Owner* указывает на владельца этого элемента, а поле *Next* — на следующий равный ему подэлемент группы. При включении видимого элемента в группу его поля изменяются так, что *Owner* содержит ссылку на экземпляр группы — владельца, а *Next* имеет значение *NIL*. После добавления к группе еще одного элемента поле *Next* ранее включенного элемента изменяет свое значение и содержит ссылку на этот новый элемент. Таким образом создается связанный список подэлементов группы (рис.17.4).

Для присоединения элемента должны использоваться методы *Insert* или *ExecView*. Метод *Insert* присоединяет очередной видимый элемент к списку подэлементов группы. В зависимости от набора связанных с подэлементом признаков этот элемент может размещаться в центре (признаки *ofCenterX* и/или *ofCenterY*), стать активным (*ofSelectable*) и, наконец, появиться на экране (*sfVisible*). После создания подэлемента управление передается следующему за обращением к *Insert* оператору программы.

Метод *ExecView* осуществляет те же действия, что и метод *Insert*, однако после создания видимого подэлемента управление будет передано в него и оператор, следующий за обращением к *ExecView*, получит управление

только после уничтожения этого подэлемента. Более точно процесс взаимодействия с программой элемента, присоединенного к группе методом *ExecView*, можно описать следующим образом. Любой видимый элемент наследует виртуальный метод *Valid*, с помощью которого он может сигнализировать своему владельцу о том, насколько успешно он выполнил возложенные на него обязанности. Обычно *Valid* возвращает *True*, если успешно создан и размещен в динамической памяти экземпляр объекта, и *False*, — в противном случае. Объект может перекрыть метод *Valid* и вернуть *False*, если он хочет оставить управление у себя, — именно так, например, поступает объект *TDialog*. Метод *ExecView* приостанавливает исполнение следующего оператора программы до тех пор, пока все подэлементы группы не вернут *Valid = True*. Таким образом, метод *ExecView* используется в том случае, когда среди подэлементов группы есть хотя бы один элемент, реализующий диалог с пользователем.

Метод *Delete* удаляет подэлемент группы из связанного списка.

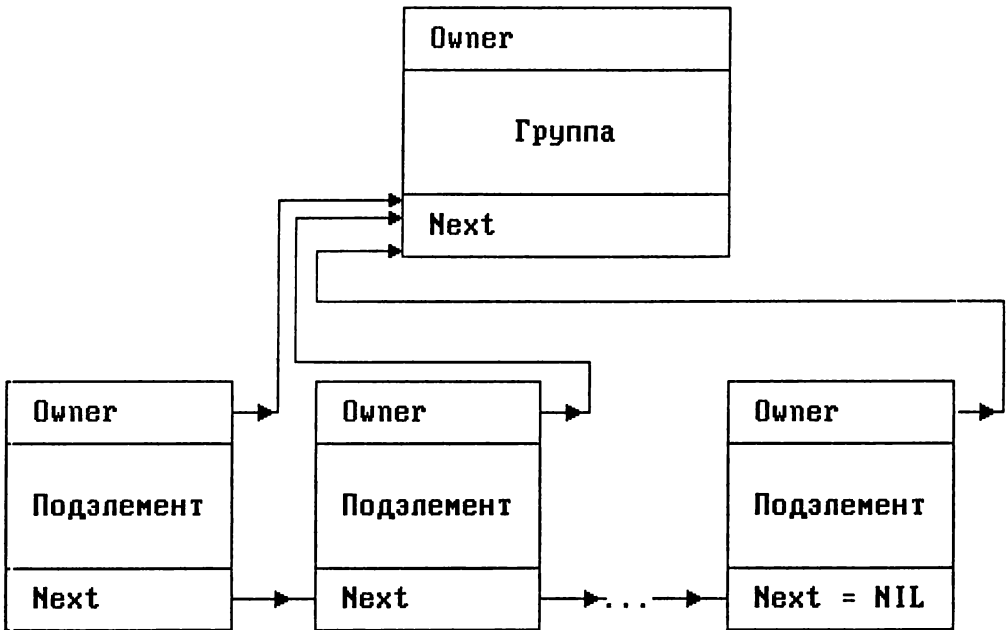


Рис. 17.4. Связанный список элементов группы

17.3.2. Z-упорядочение и дерево видимых элементов

Последовательное присоединение подэлементов к группе приводит к так называемому Z-упорядочению видимых элементов. Z-упорядочение — это трехмерная модель видимых элементов, в которой координаты *X* и *Y* определяют положение элементов на экране, а координата *Z* — порядковый номер элемента в группе. Например, на рис.17.5 показана Z-модель элементов окна с рамкой и скроллером.

В этой модели каждый элемент можно представить в виде стеклянной пластины, накладывающейся на уже существующее изображение. То

изображение, которое мы видим на экране, есть проекция трехмерной модели на плоскость XY . Трехмерный образ позволяет «взглянуть» на видимые элементы со стороны и увидеть порядок, в котором они присоединяются к группе.



Рис.17.5. Z-модель видимых элементов окна

Важно помнить, что любая группа визуализирует свои подэлементы в порядке, определяемом их Z-упорядочением. Для рис.17.5 это означает, что сначала создается изображение рамки, очерчивающей все окно, затем на рамку накладываются полосы скроллера, потом — сам скроллер с текстом, а накрывает сверху все изображение и скрепляет его в единое целое прозрачная пластина *TWindow*.

Для того чтобы группа нашла связанный с ней список подэлементов, используется поле *First*, входящее в любой видимый объект. Это поле содержит *NIL*, если данный элемент — терминальный видимый объект; если этот элемент — группа, поле *First* содержит указатель на первый вставленный в группу подэлемент, т.е. на самый «нижний» элемент в смысле Z-упорядочения. Таким образом, цепочка ссылок *First* — *Next* образует дерево видимых элементов, так как каждый элемент *Next* может быть новой группой и в этом случае его поле *First* $\neq$ *NIL*.

Программа Turbo Vision всегда владеет строкой меню, строкой статуса и рабочей зоной экрана, а следовательно, имеет дерево видимых элементов, показанное на рис.17.6.

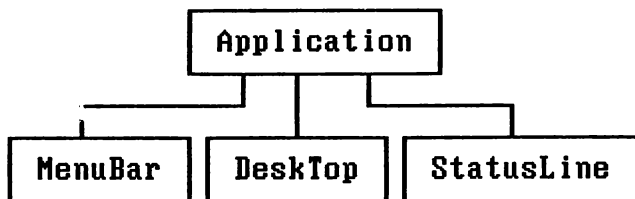


Рис.17.6. Основное дерево видимых элементов TApplication

Отметим, что деревья видимых элементов показывают принадлежность элементов, а не их иерархию в генеалогии объектов Turbo Vision, т.е. связи

на рис.17.6 определяют функциональную подчиненность экземпляров объектов *Application*, *MenuBar*, *Desktop* и *StatusLine*.

Деревья видимых элементов динамически изменяются в ходе работы программы. Они могут расти, если к программе присоединяются новые группы, или уменьшаться, если эти группы уничтожаются. В отличие от этого генеалогическое дерево объектов может только расти за счет создания потомков.

Все ветви дерева видимых элементов всегда заканчиваются терминальными видимыми объектами.

17.3.3. Активные элементы

Внутри любой группы видимых элементов в каждый момент времени может быть выбран (активизирован) один и только один элемент. Даже если в программе открыто несколько окон с текстом, активным считается только то окно, с которым Вы работаете в данный момент. Более того, поскольку окно представляет собой группу, в нем будет активным лишь один элемент. Если, например, Вы воздействуете мышью на полосу скроллера, будет активна именно эта полоса. Рис.17.7 иллюстрирует сказанное: на нем показано дерево видимых элементов для двух открытых окон, причем активные элементы выделены двойной рамкой.

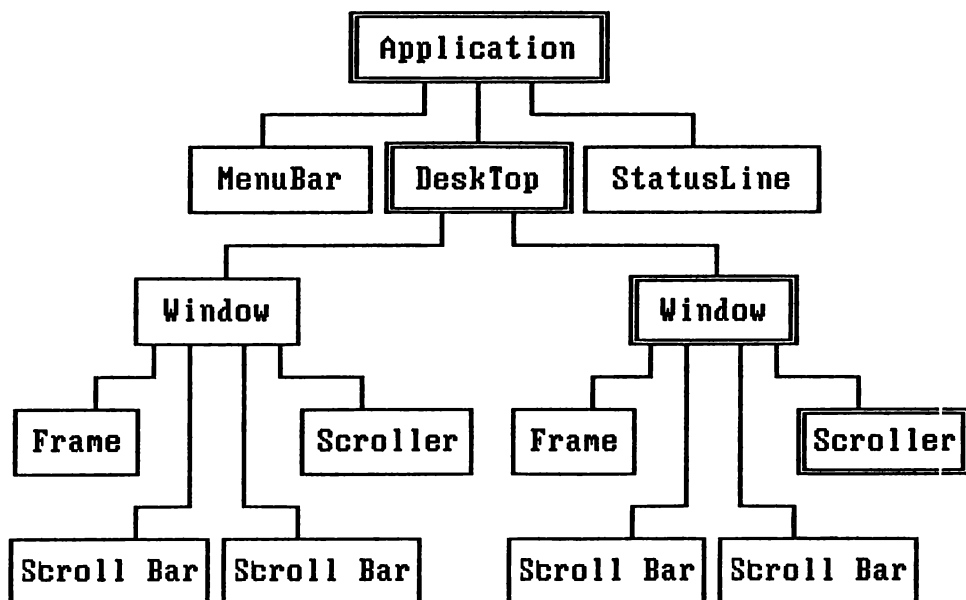


Рис.17.7. Цепочка активности видимых элементов просмотра текста

Цепочка активности видимых элементов используется при обработке событий (см. гл.18).

Активный элемент обычно выделяется на экране тем или иным способом. Например, выбранное окно очерчивается двойной рамкой, а остальные — одинарной; внутри диалогового окна активный элемент выделяется яркостью (цветом). С помощью метода *Select* видимый элемент

можно сделать активным по умолчанию в момент его создания. При активизации группы активизируется ее подэлемент, указанный как активный по умолчанию. Пользователю может потребоваться изменить текущий активный видимый элемент. Он может это сделать, манипулируя мышью, или нажав командную клавишу (если элемент связан с командной клавишей), или, наконец, с помощью клавиши *Tab*.

Заметим, что существуют видимые элементы, которые нельзя сделать активными. Например, не может быть активным видимый элемент *TBackground* (фон рабочей зоны экрана). В момент создания элемента с помощью признака *ofSelectable* Вы можете указать, будет ли этот элемент выбираемым, т.е. можно ли его сделать активным в ходе работы программы. Однако, если Вы объявите выбираемым тот же элемент *TBackground*, он все равно не сможет активизироваться, так как знает, что на самом деле активизация ему недоступна. Точно также не сможет активизироваться рамка окна (заметим, что указать на рамку мышью можно, и программа может, например, перемещать окно с рамкой, однако это еще не означает активизации рамки: рамка не может быть объектом диалога с пользователем). Обычно сброс признака *ofSelectable* используется для того, чтобы запретить элементу стать активным, он, в принципе, может активизироваться, но его активизация в программе не нужна. Таким способом можно, например, сделать неактивной метку в диалоговом окне и, следовательно, превратить ее в статический поясняющий текст.

17.4. МОДАЛЬНЫЕ ВИДИМЫЕ ЭЛЕМЕНТЫ

Все программы в среде Turbo Vision рассчитаны на диалоговый способ взаимодействия с пользователем, а следовательно, в них активно используются точки ветвления, управляемые командами пользователя (точки диалога). В точке диалога создается активный видимый элемент, называемый модальным элементом.

Примером модального элемента является диалоговое окно. Когда в программе создается и активизируется модальный элемент, только этот элемент и его подэлементы могут взаимодействовать с пользователем. Любая часть дерева видимых элементов, не являющаяся модальным элементом или не принадлежащая ему, не может быть активна в этот момент. Если, например, на экране развернуто диалоговое окно, любые отметки мышью вне его пределов или нажатие на не предусмотренные в этом окне командные клавиши будут игнорироваться.

Единственным исключением из этого правила являются командные клавиши и соответствующие поля для мыши, определенные в строке статуса. Эти клавиши (поля) всегда доступны пользователю и нажатие на них (отметка их мышью) обрабатывается модальным элементом точно также, как если бы они были определены в нем, хотя этот элемент может и не владеть строкой статуса.

Чтобы временно исключить из списка активных команду (или группу команд), определенную в строке статуса, используется метод *DisableCommands*. После завершения работы модального элемента можно восстановить активность команд методом *EnableCommands*. Параметром

обращения к этим методам служит произвольное множество типа *TCommandSet*, содержащее до 256 кодов команд. В Turbo Vision тип *TCommandSet* определен следующим образом:

type

TCommandSet = **set of** Byte;

Таким образом запрещать (и разрешать) можно только те команды, коды которых принадлежат диапазону 0...255.

Временно запрещенные команды выделяются в строке статуса пониженной яркостью (оттенком).

17.5. ИЗМЕНЕНИЕ СВОЙСТВ ЭЛЕМЕНТА

Каждый видимый элемент Turbo Vision имеет 5 полей, которые определяют его поведение в диалоговой среде и которые можно использовать для того, чтобы изменить свойства элемента. К этим полям относятся *Options*, *GrowMode*, *DragMode*, *State* и *EventMask*. Поле *EventMask* активно используется в обработчиках событий и описано в гл.18. Поля *Options*, *GrowMode* и *DragMode* доступны как для чтения, так и для записи. Поле *State* доступно только для чтения и изменить его состояние можно с помощью обращения к методу *SetState*.

17.5.1. Поле Options

Поле *Options* представляет собой шестнадцатиразрядное слово, биты (разряды) которого показаны на рис.17.8.

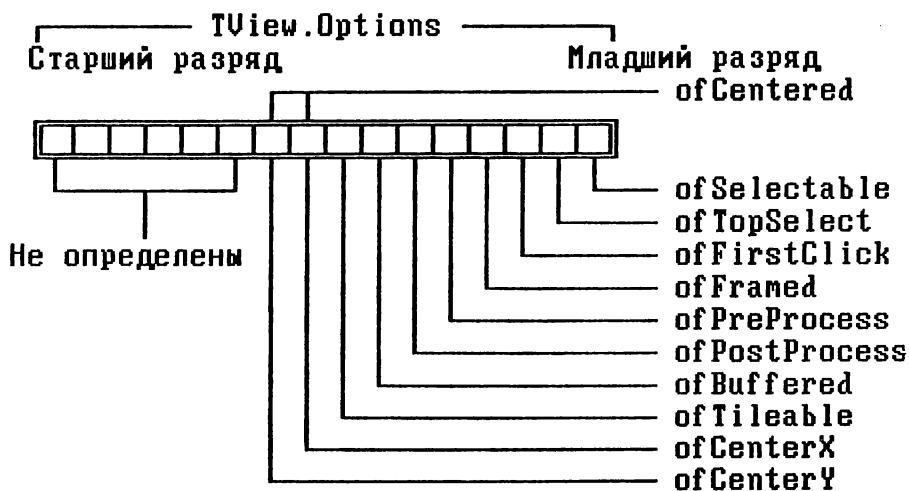


Рис.17.8. Разряды поля Options

ofSelectable

Если этот флаг установлен (имеет значение 1), пользователь может выбрать видимый элемент мышью или клавишей *Tab*. Если Вы поместили

на экран информационный элемент, Вам может потребоваться, чтобы пользователь не мог выбрать его. Например, статические текстовые объекты и рамки окон обычно не могут выбираться и имеют *ofSelectable* = 0.

ofTopSelect

Видимый элемент будет передвигаться вверх в смысле Z-упорядочения, пока не станет самым верхним среди других равных ему видимых элементов. Этот флаг в основном используется для окон в рабочей зоне экрана. Вы не должны использовать его для видимых элементов в группе.

ofFirstClick

Если флаг сброшен, отметка элемента мышью игнорируется, и наоборот — установленный флаг разрешает выбор элемента мышью. Если в диалоговое окно помещена кнопка, Вы наверняка захотите выбрать ее мышью, поэтому кнопка имеет *ofFirstClick* установленным по умолчанию. Но если Вы создаете окно, Вы можете сбросить этот флаг, если хотите, чтобы оно не откликалось на выбор мышью.

ofFramed

Если флаг установлен, видимый элемент имеет рамку.

ofPreProcess

Если флаг установлен, видимый элемент будет получать и, возможно, обрабатывать активные события до того, как их получит и обработает активный элемент. См. раздел «Фаза» в гл.18.

ofPostProcess

Установленный в 1 флаг *ofPostProcess* позволяет видимому элементу обрабатывать события после того, как они были получены активным элементом, и при условии, что активный элемент не очистил событие. См. раздел «Фаза» в гл.18.

ofBuffered

Когда этот бит установлен, образ группы при первой ее визуализации автоматически сохраняется в буфере. В следующий раз, когда группе нужно будет визуализироваться, она копирует образ из буфера на экран вместо прорисовки всех своих подэлементов, что ускоряет процесс создания изображений. Если программа испытывает недостаток в динамической памяти, монитор памяти Turbo Vision освобождает буферы группы до тех пор, пока запрос к динамической памяти не сможет быть выполнен.

Если группа имеет буфер, вызов метода *Lock* будет блокировать вывод изображения на экран до тех пор, пока не будет вызван метод *UnLock*. Сразу после вызова *UnLock* буфер группы целиком выводится на экран. Блокирование уменьшает мерцание во время сложных корректировок

экрана. Например, *TDesktop* блокирует себя, когда выводит свои подэлементы каскадом или черепицей.

ofTileable

Объект *TDesktop* может располагать связанные с ним окна каскадом или черепицей. Если окна располагаются каскадом, каждое следующее окно накладывается на предыдущее и почти полностью перекрывает его; при расположении окон черепицей каждое окно располагается так, чтобы не перекрывать другие окна. Если Вы хотите, чтобы окна располагались каскадом, Вам следует очистить флаг *ofTileable*, если черепицей — установить его в 1. Если для окна установлен режим вывода каскадом, оно будет занимать одно и то же место на экране, в то время как расположение окон черепицей приводит к изменению их размеров и положения при открытии каждого нового окна.

Расположение видимых элементов черепицей или каскадом выполняется в *TApplication.HandleEvent* очень просто:

```
cmTile:
  begin
    DeskTop^.GetExtent(R);
    DeskTop^.Tile(R);
  end;
cmCascade:
  begin
    DeskTop^.GetExtent(R);
    DeskTop^.Cascade(R);
  end;
```

ofCenterX

Этот флаг следует установить в 1, если Вы хотите, чтобы видимый элемент центрировался по горизонтали, т.е. вставлялся в группу так, чтобы слева и справа от него было приблизительно одинаковое расстояние до границ группы.

ofCenterY

Флаг обеспечивает центрирование элемента по вертикали. Если Вы хотите, чтобы Ваши окна выглядели одинаково хорошо в режимах 25 и 43/50 строк на экране, следует установить *ofCenterY* = 1.

ofCentered

Установка этих разрядов обеспечивает центровку видимого элемента относительно границ группы одновременно по горизонтальной и вертикальной осям.

17.5.2. Поле *GrowMode*

Восьмиразрядное поле *GrowMode* определяет, как должен изменяться видимый элемент, когда его владелец (группа) изменяет свои размеры. Разряды этого поля показаны на рис.17.9.

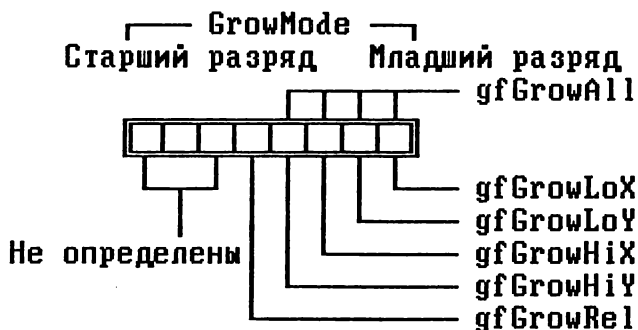


Рис.17.9. Разряды поля *GrowMode*

Среда Turbo Vision позволяет изменять размеры окон с помощью мыши: для этого надо «схватить» мышью правый нижний угол окна. Флаги *GrowMode* определяют, как будут вести себя в этой ситуации вставленные в окно элементы. Напомню, что изображение встроенного элемента всегда отсекается границами группы. Стандартное состояние элементов среды Турбо Паскаль соответствует установленным флагам *gfGrowHiX* и *gfGrowHiY*.

gfGrowLoX

Если флаг установлен, левая граница видимого элемента всегда располагается на одинаковом расстоянии от правой границы группы-владельца. Таким образом, при уменьшении горизонтального размера скна вставленный в него видимый элемент смещается влево и, возможно, отсекается левой границей окна.

gfGrowLoY

Если флаг установлен, верхняя граница видимого элемента всегда располагается на одинаковом расстоянии от нижней границы группы, т.е. уменьшение вертикального размера окна приводит к смещению элемента вверх.

gfGrowHiX

Если флаг установлен, правая граница видимого элемента всегда располагается на одинаковом расстоянии от левой границы группы, т.е. при уменьшении горизонтального размера окна видимое на экране положение его внутреннего элемента остается неизменным и, возможно, отсекается правой границей окна.

gfGrowHiY

Если флаг установлен, нижняя граница видимого элемента всегда располагается на одинаковом расстоянии от верхней границы группы, т.е. при уменьшении вертикального размера окна видимое на экране положение его внутреннего элемента останется неизменным. Стандартное состояние элементов среды Турбо Паскаль соответствует установленным флагам *gfGrowHiX* и *gfGrowHiY*.

gfGrowAll

Если разряды *gfGrowAll* установлены в 1, видимый элемент передвигается в процессе изменения размеров своего владельца, отсекаясь его левой и верхней границами.

gfGrowRel

Если флаг установлен, видимый элемент пропорционально изменяет свои размеры при изменении размеров владельца. Вы должны использовать эту опцию только с *TWindow*, или с наследниками от *TWindow*, которые присоединяются к панели экрана. В этом случае окна сохраняют свои относительные размеры и положения на экране при переходе от 25 строк к 43/50 строкам и обратно. Этот флаг не предназначен для использования с видимыми элементами внутри окна.

17.5.3. Поле *DragMode*

Поле *DragMode* размером в один байт определяет, как ведет себя видимый элемент при перемещении владельца. Напомню, что Turbo Vision позволяет перемещать окно на экране с помощью мыши, если «схватить» мышью верхнюю рамку окна.

Должен заметить, что мне не удалось добиться сколько-нибудь заметных результатов при различных установках поля *DragMode*. По умолчанию Turbo Vision устанавливает *DragMode* = 32 = *dmLimitLoY*.

На рис.17.10 указаны разряды поля *DragMode*.

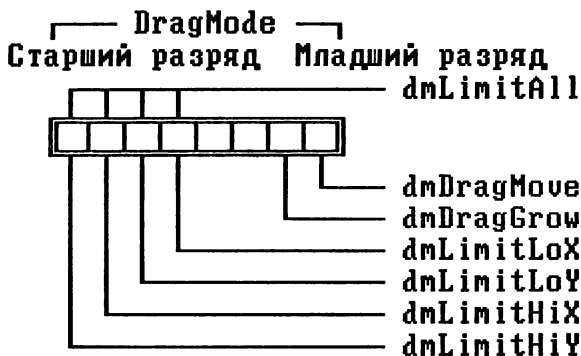


Рис.17.10. Разряды поля *Drag*

dmDragMove

Установленный флаг *dmDragMode* разрешает перемещать окно с помощью мыши.

dmDragGrow

Если флаг установлен, окно может изменять свои размеры.

dmLimitLoX

Если флаг установлен, левая сторона видимого элемента не может выходить за границы своего владельца.

dmLimitLoY

Если флаг установлен, верхняя часть видимого элемента не может выходить за границы своего владельца.

dmLimitHiX

Если флаг установлен, правая сторона видимого элемента не может выходить за границы своего владельца.

dmLimitHiY

Если флаг установлен, нижняя часть видимого элемента не может выходить за границы своего владельца.

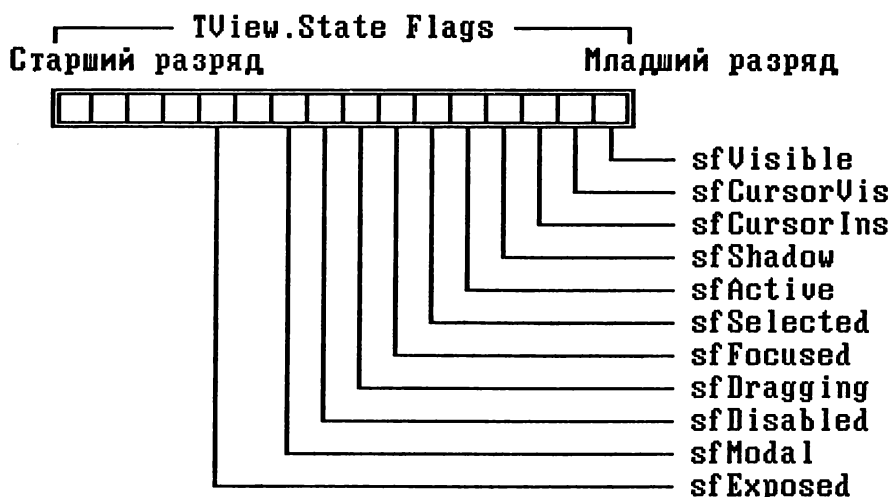
dmLimitAll

Если флаг установлен, ни одна часть видимого элемента не может выходить за границы своего владельца.

17.5.4. Поле State

Шестнадцатиразрядное поле *State* хранит различные признаки, определяющие поведение или состояние видимого элемента.

На рис.17.11 показаны разряды поля *State*.

Рис.17.11. Разряды поля *State**sfVisible*

Разрешает визуализировать видимый элемент, если визуализируется его владелец. По умолчанию этот флаг установлен.

sfCursorVis

Разрешает показать курсор внутри видимого элемента. По умолчанию этот флаг очищен.

sfCursorIns

Если этот флаг установлен, курсор будет занимать всю высоту строки, если сброшен, курсор имеет стандартную высоту (две нижних строки развертки).

sfShadow

Если флаг установлен, видимый элемент имеет «тень».

sfActive

Указывает, является ли данный элемент активным окном или подэлементом активного окна.

sfSelected

Указывает, является ли данный элемент текущим (выбранным) видимым элементом. Каждый объект *TGroup* имеет поле *Current*, содержащее указатель на текущий выбранный подэлемент или *NIL*, если не выбран ни один подэлемент. В каждый момент времени в группе может быть выбран только один подэлемент.

sfFocused

Указывает, принадлежит ли данный элемент цепочке активности видимых элементов, т.е. выбран ли он и все его владельцы в данный момент времени.

sfDragging

Разрешает изменять размеры элемента.

sfDisable

Флаг запрещает выбирать данный видимый элемент. Если флаг сброшен, элемент можно выбрать мышью или клавишей *Tab*.

sfModal

Если флаг установлен, данный элемент — это модальный элемент. В программе Turbo Vision всегда имеется один и только один модальный элемент. Обычно это экземпляр *TApplication* или *TDialog*. Модальный видимый элемент активизируется вызовом метода *ExecView* и образует корень активного дерева событий: он перехватывает события и управляет ими до тех пор, пока не будет вызван его метод *EndModal* (см. гл.18). Модальный элемент может передавать события своим подэлементам и получать события от них, но он никогда не передает события своему владельцу (события локализуются в модальном элементе).

sfExposed

Указывает, виден ли элемент или любая его часть в данный момент времени (элемент может закрываться другими видимыми элементами). Если флаг установлен, метод *Exposed* данного элемента возвращает значение *True*.

17.5.5 Воздействие на состояние поля *State*

В отличие от других полей поле *State* доступно только для чтения (поля *Options*, *DragMode* и *GrowMode* доступны также и для записи). Это означает, что в программе не может использоваться оператор вида

```
State := NewState;
```

Установить новое состояние этого поля можно только с помощью метода *SetState*, доступного любому видимому элементу. Метод *SetState* объявлен в Turbo Vision следующим образом:

```
type
  TView = object (TObject)
  Procedure SetState(AState: Word; Enable: Boolean); Virtual;
  .....
end;
```

При обращении к методу параметр *AState* должен содержать маску разрядов поля *State*, а признак *Enable* указывает, должны ли устанавливаться эти разряды (*Enable = True*) или очищаться (*Enable = False*).

Отметим, что Turbo Vision вызывает метод *SetState* всякий раз, когда элемент активизируется или выбирается. Это дает возможность программе устанавливать новое состояние элемента или реагировать на действия пользователя (например, на активизацию видимого элемента). В отличие от этого поля *Options*, *DragMode* и *GrowMode* активизируются только при инициации элемента и далее обычно не меняются.

Видимые элементы часто перекрывают метод *SetState*, чтобы гибко реагировать на действия пользователя. Например кнопка (объект *TButton*) просматривает флаги поля *State* и изменяет свой цвет на бирюзовый, когда становится активной. В следующем примере объект *TButton* вначале проверяет, принадлежит ли он активному окну, чтобы решить, рисовать себя или нет. Затем он проверяет, выбрана ли кнопка в окне и, если это так, вызывает свой метод *MakeDefault*, чтобы установить или сбросить (в зависимости от *Enable*) свою активность по умолчанию:

```
Procedure TButton.SetState(AState: Word; Enable: Boolean);
begin
  TView.SetState(AState, Enable);
  if (AState and (sfSelected + sfActive)) <> 0 then
    DrawView;
  if (AState and sfFocused) <> 0 then
    MakeDefault(Enable);
end;
```

Другой пример: допустим, что Ваша программа включает в себя текстовый редактор и Вы хотите разрешить или запретить все команды редактирования в полосе меню в зависимости от того, открыт редактор или нет. В этом случае Вы можете создать такой метод *SetState* для текстового редактора:

```
Procedure TEditor.SetState(AState: Word; Enable: Boolean);
const
  EditorCommands = [cmSearch, cmReplace, cmSearchAgain,
                    cmGotoLine, cmFindProc, cmFindError];
begin
  TView.SetState(AState, Enable);
  if AState and sfActive <> 0 then
    if Enable then
      EnableCommands(EditorCommands)
    else
      DisableCommands(EditorCommands);
end;
```

Глава 18

СОБЫТИЯ

Как уже неоднократно подчеркивалось, программы, работающие в среде Turbo Vision, — это программы, управляемые событиями. В этой главе подробно рассматривается механизм событий и способы их использования.

18.1. ПРИРОДА СОБЫТИЙ

События представляют собой небольшие пакеты информации, которыми обмениваются видимые элементы и которые создаются средой Turbo Vision в ответ на те или иные действия пользователя. Нажатие на любую клавишу или манипуляция мышью порождает событие, которое передается по цепочке активности видимых элементов до тех пор, пока не найдется элемент, знающий как обработать это событие. Может оказаться, что в программе нет ни одного элемента, способного обработать событие. В этом случае обычно ничего не происходит, по умолчанию Turbo Vision просто удаляет ненужное событие, однако программист может предусмотреть свою реакцию в этой ситуации.

Важно помнить, что события сами по себе не производят никаких действий в программе, но в ответ на событие могут создаваться новые видимые элементы, модифицироваться или уничтожаться существующие элементы, что и приводит к изменению содержимого экрана. Иными словами, все действия по созданию и изменению изображения реализуются видимыми объектами, а события лишь управляют их работой — именно это мы имеем в виду, говоря о программах, управляемых событиями.

Технически событие представляет собой обычную для Турбо Паскаля запись, имеющую следующую структуру:

type

TEvent = record

What: Word;

{Тип события}

case Word of

evNothing: ();

{Пустое событие}

evMouse: (

{Событие от мыши:}

Buttons: Byte;

{Состояние кнопок}

Double : Boolean;

{Признак двойного нажатия}

Where : TPoint);

{Координаты мыши}

evKeyDown: (

{Событие от клавиатуры:}

case Integer of

0: (KeyCode: Word);

{Код клавиши}

1: (CharCode: Char;

ScanCode: Byte));

evMessage: (

{Событие-сообщение}

Command: Word;

{Код команды}

```

case Word of
  0: (InfoPtr : Pointer);
  1: (InfoLong: LongInt);
  2: (InfoWord: Word);
  3: (InfoInt : Integer);
  4: (InfoByte: Byte);
  5: (InfoChar: Char));
end;

```

Ядром события является поле *What*, которое описывает тип события. Оставшаяся часть записи содержит дополнительную информацию, например, код нажатой клавиши или координаты точки, в которой была нажата кнопка мыши.

18.2. ВИДЫ СОБЫТИЙ

Существуют 4 основных класса событий: события от мыши, события от клавиатуры, сообщения и пустые события. Внутри класса события могут подразделяться на виды. Например, класс событий от мыши состоит из таких видов, как перемещение указателя мыши, нажатие на кнопку мыши и т.п. Каждый класс имеет определенную маску, с помощью которой можно легко идентифицировать класс события, не анализируя его вид. Например, вместо того, чтобы проверять 4 различных вида событий от мыши, можно просто проверить, пришло ли это событие от мыши или оно имеет другой источник. Два следующих оператора идентичны:

```

if Event.What and ((svMouseDown or evMouseDown or
  evMouseMove or evMouseAuto) <> 0) then ...
if Event.What and (evMouse <> 0) then ...

```

Для анализа класса событий используются следующие маски:

```

evNothing    {'Пустое' событие};
evMouse      {Событие от мыши};
evKeyboard   {Событие от клавиатуры};
evMessage    {Сообщение}

```

На рис.18.1 показаны разряды поля *What* и соответствующие маски.

18.2.1. События от мыши

Существуют 4 вида событий от мыши: событие *evMouseDown* возникает как отклик на нажатие кнопки мыши; когда кнопка отпускается, возникает событие *evMouseUp*; перемещение указателя мыши порождает событие *evMouseMove*; наконец, если кнопка мыши удерживается в нажатом состоянии, Turbo Vision периодически генерирует событие *evMouseAuto*. С каждым событием от мыши в записи *TEvent* передаются также координаты, которые имеет указатель мыши в данный момент. Отметим, что в отличие от других координат Turbo Vision координаты мыши задаются относительно границ экрана, а не относительно границ какого-либо видимого элемента. Горизонтальная координата мыши меняется в диапазоне

от 0 до 79, вертикальная — от 0 до 24 (режим 25 строк на экране) или от 0 до 42/49 (режим 43/50 строк).

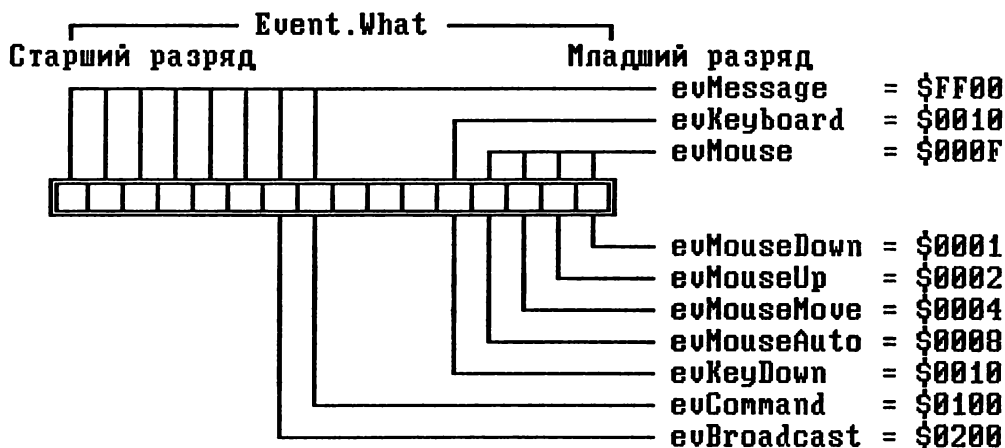


Рис.18.1. Разряды поля What

18.2.2. События от клавиатуры

В этом классе событий имеется единственное событие *evKeyDown*, связанное с нажатием на клавишу. В поле *TEvent.KeyCode* в этом случае содержится так называемый расширенный код клавиши, который Вы можете интерпретировать как совокупность двух байт: *CharCode* и *ScanCode*. При нажатии на алфавитно-цифровую клавишу поле *CharCode* содержит соответствующий ASCII-символ, при нажатии на функциональную клавишу поле *CharCode* содержит символ #0, а поле *ScanCode* — код сканирования нажатой клавиши. Для облегчения идентификации нажатой клавиши можно использовать константы *kbXXXX*, определенные в интерфейсной части модуля *Drivers*.

18.2.3. Сообщения

Сообщения бывают трех видов: команды, общие сообщения и пользовательские сообщения. Команды помечаются в поле *What* флагом *evCommand*, общие сообщения — флагом *evBroadcast* и пользовательские сообщения — константой, определенной пользователем. Большинство событий преобразуется в команды. Например, если пользователь отметит мышью какое-то поле строки статуса, сообщение от мыши поступит в конечном счете в обработчик событий строки статуса (любой видимый элемент имеет метод *HandleEvent*, называемый обработчиком событий, см. п.16.6.2), который определит, какое именно поле было отмечено. С каждым полем строки статуса обычно связана какая-то команда, поэтому обработчик очистит пришедшее к нему сообщение от мыши и создаст новое сообщение, содержащее выбранную команду. Общие и пользовательские сообщения не являются исключением и обычно также преобразуются в команды.

18.2.4. Пустые события

Пустым событие становится после его обработки каким-либо видимым элементом. Технически пустое событие соответствует значению *TEvent.What = evNothing = 0*. Когда событие обработано, видимый элемент вызывает метод *ClearEvent*, с помощью которого в поле *What* устанавливается значение *evNothing*. Объекты должны игнорировать событие *evNothing*, поскольку оно уже обработано каким-то видимым элементом.

18.3. МАРШРУТИЗАЦИЯ СОБЫТИЙ

Как уже говорилось, любая программа, работающая в среде Turbo Vision, является прямым или косвенным (через *TApplication*) потомком *TProgram* и основана на обращении к трем главным методам этого объекта: *Init*, *Run* и *Done*. Например:

```
Uses App, ...;
type
  TMyProgram = object (TApplication)
    .....
  end;
var
  MyProgram = TMyProgram;
  .....
begin
    {Исполняемая часть программы:}
  MyProgram.Init;      {Инициировать программу}
  MyProgram.Run;       {Выполнить программу}
  MyProgram.Done       {Завершить работу}
end.
```

Процесс получения и обработки событий иницируется методом *TProgram.Run*, который для этого обращается к методу *TGroup.Execute* (любая программа является потомком *TGroup*). Метод *TGroup.Execute* реализует следующий цикл:

```
var
  Event: TEvent;
  .....
begin
  Event.What := evNothing; {Инициировать пустое событие}
  repeat {Основной цикл программы}
    if Event.What <> evNothing then
      EventError(Event); {Событие не очищено - ошибка}
      GetEvent(Event); {Получить событие}
      HandleEvent(Event); {Передать событие обработчику}
    until EndState <> Continue;
    .....
end;
```


Метод *GetEvent* наследуется всеми видимыми элементами от *TView* и является основным источником событий. Этот метод вначале проверяет, не подготовил ли событие метод *PutEvent* и, если это так, возвращает его. Затем *GetEvent* проверяет клавиатуру и мышь. Если какое-то из этих устройств изменило свое состояние, метод формирует и возвращает соответствующее событие. Если изменения не произошло, *GetEvent* обращается к методу *TProgram.Idle*, чтобы запустить «фоновую» задачу (подробнее об этом см. п.18.6).

С помощью вызова метода *TGroup.Execute* события всегда начинают свой путь с модального видимого элемента. Напомню, что модальный элемент определяет точку диалога; в программе в каждый момент времени есть один и только один модальный элемент (см. п.17.4). В самом начале программы таким элементом является обычно экземпляр объекта *TProgram* или его потомка. Если в программе открыто модальное диалоговое окно, оно обратится к *TGroup.Execute* и, следовательно, путь событий начнется с этого окна. В любом случае начинает обработку события метод *HandleEvent* модального видимого элемента. Дальнейший маршрут события зависит от того, является ли событие позиционированным, активным или общим.

18.3.1. Позиционированные события

Позиционированные события — это всегда события от мыши (*evMouse*). Модальный видимый элемент получает позиционированное событие первым и начинает просматривать свои подэлементы в порядке, обратном их Z-упорядочению (см. п.17.3.2), до тех пор, пока не найдет подэлемент, координаты которого включают в себя координаты указателя мыши. Затем модальный элемент передает событие найденному подэлементу. Поскольку видимые элементы часто перекрываются на экране, может оказаться, что координата указателя мыши принадлежит более чем одному видимому элементу. Следование обратному Z-упорядочению гарантирует, что событие получит самый верхний видимый элемент. Заметим, что Вам не нужно проследивать Z-упорядочение элементов в Вашем обработчике событий: достаточно вызвать унаследованный метод *HandleEvent*, который автоматически направит событие нужным образом.

Процесс передачи события продолжается до тех пор, пока не обнаружится терминальный видимый элемент (например, полоса скроллинга) или не будет найден ни один подэлемент с нужными координатами элемент не знает, как обработать событие, он передает его вверх по активной цепочке своему владельцу. Технически это реализуется с помощью выхода из обработчика *HandleEvent* вызовом *Exit*; событие не следует очищать обращением к *ClearEvent*. Процесс повторяется до тех пор, пока событие не будет обработано или не достигнет модального элемента. Если модальный элемент не знает, как обработать вернувшееся к нему событие, он вызывает метод *EventError*.

В Turbo Vision существуют средства, позволяющие видимым элементам, не принадлежащим цепочке активности, получать и обрабатывать активные события (см. п.18.4).

18.3.2. Общие события

Общие события — это общие сообщения или пользовательские сообщения. Общие события не имеют конкретного адресата и посылаются всем подэлементам текущего модального элемента.

Модальный элемент получает событие и начинает передавать его своим подэлементам в Z-порядке. Если какой-то из подэлементов — группа, он также передает событие своим подэлементам и также в Z-порядке. Процесс продолжается до тех пор, пока не будет найден элемент, который обработает событие, или пока событие не получат все видимые элементы, принадлежащие (прямо или косвенно) модальному элементу.

Общие события обычно используются для организации межэлементных связей. Подробнее об этом см. п.18.7.

18.4. ФАЗА СОБЫТИЙ

Обычно активные события (*evKeyDown* и *evCommand*) получают и обрабатывают видимые элементы, принадлежащие цепочке активности. Однако часто возникают ситуации, когда необходимо, чтобы активное событие обработал неактивный элемент. Например, если на экране активно окно скроллера с полосами скроллинга, то события от клавиатуры будут передаваться окну. Как заставить в этом случае полосы реагировать на нажатие клавиш *PgUp* или *PgDn*? Для этого в Turbo Vision предусмотрен специальный механизм, основанный на так называемой фазе события. Когда модальный элемент получает событие, его передача выполняется в следующей последовательности:

- событие посылается в Z-порядке всем видимым элементам, которые принадлежат модальному элементу и у которых поле *Options* имеет установленный флаг *ofPreProcess*;
- если событие не очищено ни одним из них, оно посылается активным элементам (по цепочке активности);
- если событие все еще не очищено, оно посылается в Z-порядке всем видимым элементам, у которых установлен флаг *ofPostProcess*.

Таким образом, Вы должны установить флаги *ofPreProcess* или *ofPostProcess* (или оба вместе) при инициации видимого элемента, если хотите, чтобы он мог получить активное событие до или после (или и до и после) того, как его получают активные элементы.

Для предыдущего примера необходимо инициировать полосы скроллинга с установленными флагами *ofPostProcess*, если требуется, чтобы полосы «увидели» и обработали нажатие на клавиши смещения курсора, *PgUp*, *PgDn* и т.д. Разумеется, в этом случае полосы получают событие *evKeyDown* только при условии, что скроллер сам не обработает это событие.

В некоторых ситуациях элемент, перехватывающий событие и до, и после активных элементов, должен модифицировать свое поведение в зависимости от фазы события. Рассмотрим такой типичный пример. Пусть в программе создано диалоговое окно, имеющее строку ввода и три кнопки, для которых определены командные клавиши *Q*, *W*, *E*. Как добиться того, чтобы эти клавиши использовались в качестве командных клавиш, т.е.

приводили к «нажатию» соответствующих кнопок при условии, что активна любая кнопка, а в сочетании с клавишей *Alt*, — если активна строка ввода (именно так используются командные клавиши в диалоговом окне среды Турбо Паскаля)? Если инициировать кнопки с флагом *ofPreProcess*, они смогут без труда определить факт нажатия на командную клавишу, однако в строке ввода пользователь не сможет ввести буквы *Q*, *W* и *E*, так как они будут перехвачены кнопками до того, как событие от клавиши получит строка ввода. Если инициировать кнопки с флагом *ofPostProcess*, пользователь не сможет использовать сочетания *Alt*—<клавиша> для нажатия кнопки, если активна строка ввода: все события *evKeyDown* будут в этом случае направляться в строку. Решение очевидно: нужно определить оба флага, но на препроцессорной фазе следует проверять ввод *Alt*—<клавиша>, а на постпроцессорной — <клавиша>.

Для реализации этих проверок обработчик событий объекта *TButton* должен каким-то образом определить текущую фазу события. С этой целью в любой группе предусмотрено поле *Phase*. Это поле доступно только для чтения и содержит одно из значений *phPreProcess*, *phFocused* или *phPostProcess* в зависимости от фазы события.

Следующий фрагмент иллюстрирует ту часть обработчика событий кнопок, которая проверяет командную клавишу:

```
evKeyDown: {Это часть оператора case}
begin
  C := HotKey(Title^);      {Получаем в C букву клавиши}
  {Проверяем Alt-<клавиша>:}
  if (Event.KeyCode = GetAltCode(C)) or
    {Проверяем <клавиша>:}
    (Owner^.Phase = phPostProcess) and (C <> #0) and
    (UpCase(Event.CharCode) = C) or
    {Проверяем активность и нажатие пробела:}
    (State and sfFocused <> 0) and (Event.CharCode
  == ' ') then
    Press {Да, кнопка выбрана: выдаем нужную команду}
end;
```

В этом фрагменте не показанная здесь функция *HotKey* выделяет из надписи на кнопке символ командной клавиши (он обрамляется символом «~»), а стандартная для Turbo Vision функция *GetAltCode* преобразует этот символ в расширенный код клавиш *Alt*—<клавиша>. Метод *TButton.Press* реализует «нажатие» на кнопку и выдает сообщение *evBroadcast* с командой *TButton.Command*.

Отметим, что рассмотренный пример приведен только в качестве иллюстрации: стандартный объект *TButton* реализует свой обработчик событий именно таким образом и Вам нет нужды переопределять его (по умолчанию экземпляр *TButton* иницируется с установленными флагами *ofPreProcess* и *ofPostProcess*).

18.5. КОМАНДЫ

Поскольку события обычно связаны с какими-то действиями пользователя программа должна, как правило, реагировать на эти действия изменением видимого изображения. С точки зрения Turbo Vision это означает, что обработчики событий должны преобразовывать события в действия, связанные с реакцией на произошедшее событие. Для реализации этих действий в Turbo Vision предусмотрены команды.

Команда — это просто целое число без знака, кодирующее определенную последовательность действий. В Turbo Vision предусмотрен ряд стандартных команд для реализации многих стандартных действий. Например, команда *cmQuit* реализует завершение работы программы и выход в ДОС, *cmClose* закрывает активное окно, *cmZoom* распаковывает окно на весь экран или возвращает ему прежние размеры и т.д. Идентификаторы *cmXXXX* являются идентификаторами предопределенных констант, которые кодируют стандартные действия (например, *cmQuit* = 1, *cmZoom* = 5 и т.д.).

В своей программе Вы можете наряду со стандартными командами определить и использовать собственные команды для реализации специфических действий. Для этого необходимо создать свою константу-команду и в нужный момент сообщить видимым элементам о необходимости выполнить ее. Разумеется, Вы можете использовать произвольный идентификатор для вновь определяемой команды, однако Ваша программа станет намного понятнее, если при образовании новых идентификаторов Вы будете следовать каким-то правилам. В этом смысле использование префикса *cm* в идентификаторах новых команд кажется вполне логичным.

18.5.1. Преобразование активных событий в команды

Как указать на необходимость выполнения команды? Для этого в Turbo Vision Вы просто создаете событие-команду (*evCommand*), в поле *Command* которой помещаете код нужной команды. Например:

const

```
cmMyCommand = 100;
```

```
.....
```

{Ниже показан фрагмент обработчика событий:}

```
Event.What := evCommand;      {Определяем событие-команду}
Event.Command := cmMyCommand; {Указываем код команды}
Event.InfoPtr := NIL;          {Признак активного события}
PutEvent(Event);               {Создаем событие}
```

В этом фрагменте событие-команда создается обращением к методу *PutEvent*. Заметим, что поле *Event.InfoPtr* должно содержать *NIL*, если событие активно; если событие уже обработано, для его очистки используется стандартный метод *ClearEvent*, который помещает в поле *What* признак *evNothing*, а в поле *InfoPtr* — указатель *@Self* (указатель на таблицу виртуальных методов объекта). Подробнее об использовании поля *Event.InfoPtr* см. п. 18.7.

Вновь созданное событие вернется модальному элементу, который должен знать, как его обрабатывать. Например, часто событие — команда создается обработчиком событий строки статуса как реакция на нажатие предусмотренной в этой строке командной клавиши или отметку мышью соответствующего поля. В этом случае обработчик событий программы может иметь такой вид:

```

Procedure MyProgram.HandleEvent(var Event);
.....
begin
  Inherited HandleEvent(Event);
  case Event.What of
    evCommand:
      begin                                {Обработать команды;}
        case Event.Command of
          cmMyCommand: MyProcedure; {Выполнить действия,
                                     связанные с командой cmMyCommand}
          .....
        else
          exit {Не обрабатывать непредусмотренные команды}
        end; {case}
      ClearEvent(Event) {Очистить событие}
    end;
    .....
end;

```

Часто модальным элементом является диалоговое окно с несколькими кнопками. Если Вам необходимо связать с кнопками свои команды и получить выбранную в диалоге команду, Вы можете закрыть окно с помощью вызова *EndModal*, передав этому методу в качестве параметра код команды.

В следующем примере создается диалоговое окно с двумя кнопками. При нажатии кнопки «Команда *cmPrint*» окно закроется и на экран будет выведена строка

Действие команды *cmPrint*

Если нажать кнопку «Выход» или закрыть окно клавишей *Esc*, эта строка не появится.

Uses CRT, App, Dialogs, Objects, Drivers, Views;

type

 PProg = ^TProg;

 TProg = **object** (TApplication)

Constructor Init;

end;

 PDial = ^TDial;

 TDial = **object** (TDialog)

Procedure HandleEvent(**var** Event: TEvent); **Virtual**;

end;

```

const
    cmPrint = 100;
Constructor TProg.Init;
var
    R: TRect;
    Dia: PDial;
begin
    Inherited Init;
    R.Assign(20,9,60,17);
    Dia := New(PDial, Init(R, ''));
    R.Assign(3,4,22,6);
    Dia^.Insert(New(PButton,
        Init(R, 'Команда cm~P~rint', cmPrint, bfDefault)));
    R.Assign(23,4,35,6);
    Dia^.Insert(New(PButton, Init(R, 'Выход', cmCancel, bfNormal)));
    if ExecView(Dia) = cmPrint then
        begin
            {Вывод сообщения "в лоб", с помощью стандартных средств Турбо
            Паскаля. В TurboVision есть более удобные способы вывода сооб-
            щений}
            GotoXY(30,12);
            TextColor(Black);
            TextBackground(White);
            Write(' Действие команды cmPrint ')
        end
    end; {TProg.Init}
Procedure TDial.HandleEvent(var Event: TEvent);
begin
    Inherited HandleEvent(Event);
    if (Event.What = evCommand) and
        (Event.Command = cmPrint) then EndModal(cmPrint)
end; {TDial.HandleEvent}
var
    Prog: TProg;
begin
    Prog.Init;
    Prog.Run;
    Prog.Done
end.

```

В обработчике событий диалогового окна *TDial.HandleEvent* вначале вызывается стандартный обработчик *TDial.HandleEvent*. Это дает возможность кнопке «Команда cmPrint» преобразовать событие, связанное с ее выбором, в команду *cmPrint*. Вновь созданное событие возвращается обработчику *TDial.HandleEvent*, т.к. именно он является обработчиком событий модального элемента. Возвращаемая модальным элементом команда служит значением стандартной функции *ExecView*. Для упрощения программы вывод сообщения реализуется стандартными средствами Турбо

Паскаля. В Turbo Vision имеется процедура *MessageBox*, обеспечивающая более удобный вывод сообщений.

18.5.2. Запрещение и разрешение команд

В качестве значения константы — команды можно использовать любое число в диапазоне от 0 до 65535, однако следует учесть, что диапазоны 0...99 и 256...999 Turbo Vision резервирует для стандартных команд и их не следует использовать для определения команд пользователя. Два диапазона зарезервированных команд выбраны потому, что команды с кодами 0...255 можно временно запретить, в то время как остальные команды запретить невозможно. Для запрещения или разрешения команд используется глобальный тип *TCommandSet*, представляющий собой множество чисел в диапазоне 0...256 (мощность множеств в Турбо Паскале не может превышать 256, вот почему запретить можно только первые 256 команд). Команды запрещаются обращением к методу *DisableCommands*, а разрешаются обращением к *EnableCommands*. Диалоговые элементы, связанные с запрещенными командами, выделяются оттенком и их нельзя выбрать мышью или командными клавишами. Например, если в конструкторе *TProg.Init* (см. предыдущий пример) перед оператором

```
if ExecView(Dia) = cmPrint then
```

вставить оператор

```
DisableCommand([cmPrint]);
```

кнопка «Команда cmPrint» будет выведена цветом фона окна и станет недоступна для диалога.

18.6. МОДИФИКАЦИЯ И ОБРАБОТКА СОБЫТИЙ

18.6.1. События, определенные пользователем

Старшие разряды поля *Event.What* используются для указания на то, что событие относится к классу сообщений. Первые шесть разрядов в этом поле программист может использовать для определения собственных классов событий.

Необходимость в новых классах событий может возникнуть в том случае, когда Ваша программа работает с нестандартными источниками информации. Если, например, Вы собираетесь работать с последовательным портом, Вы, возможно, захотите определить класс событий *evSerial*, используя для его идентификации один или несколько старших разрядов поля *Event.What*. Технически получить информацию из нового источника и сделать ее событием можно путем перекрытия метода *TProgram.GetEvent* (см.п.18.6.4) или «фонового» метода *TProgram.Idle* (см.п.18.6.5).

По умолчанию все новые классы событий маскируются маской *evMessage*, т.е. считаются сообщениями. Такие события модальный элемент рассылает всем своим подэлементам в Z-порядке (см.17.3.2). Если Вы

хотите, чтобы новый класс событий передавался как позиционированные или активные события, Вы можете изменить маски этих событий. В Turbo Vision определены маски *PositionalEvents* и *FocusedEvents*. Первая позволяет всем видимым элементам рассматривать событие как позиционированное, вторая — как активное. По умолчанию маска *PositionalEvents* выделяет все биты *evMouse*, а *FocusedEvents* содержит *evKeyboard*. Если Вам понадобится, чтобы новый класс событий обрабатывался так же как активные или позиционированные события, Вам необходимо добавить к нужной маске биты, соответствующие этому классу в поле *What*.

Например, создан новый класс событий *evSerial* с маской \$8000, т.е. этот класс связан со старшим разрядом поля *What*. Если потребуется обработать этот класс так же как активные события, нужно задать новую маску:

```
const
  evSerial = $8000;
.....
FocusedEvents := FocusedEvents or evSerial;
```

Обратите внимание: при добавлении к любой маске новых разрядов следует применять поразрядные операции над целыми числами (операции *or*, *and*, *not*). Не следует использовать операцию арифметического сложения (+), так как в этом случае разрешен поразрядный перенос и вновь полученная маска может оказаться не той, какую Вы хотели. Если, например, к маске *evMouse* прибавить единицу, получится маска *evKeyboard*, т.е.

$$\text{evKeyboard} = \text{evMouse} + 1$$

В то же время операция

$$\text{evMouse or } 1$$

не вызовет поразрядного переноса и маска *evMouse* останется прежней, т.к. ее младший бит уже установлен.

18.6.2. Маскирование и очистка событий

Каждый видимый элемент имеет поле *EventMask*. По умолчанию значение этого поля устанавливается таким образом, чтобы видимый элемент обрабатывал все необходимые ему события и не откликался на другие. Например, *TProgam* и *TDialog* имеют *EventMask* = \$FFFF, что позволяет им откликаться на любые возможные события, в том числе и определенные программистом. Кнопка *TButton* имеет *EventMask* = \$0311, т.е. откликается на события *evBroadcast*, *evCommand*, *evKeyDown*, *evMouseAuto*, *evMouseUp* и *evMouseDown*. Как видим, ей доступны все стандартные события, кроме *evMouseMove* — кнопка не может реагировать на перемещение мыши.

Соответствующей установкой поля *EventMask* Вы можете запретить или разрешить любому видимому элементу реагировать на те или иные события. Например, Вы можете создать кнопку, доступ к которой возможен только с помощью мыши:

var

MouseButton: PButton;

.....

MouseButton := New(PButton, Init(...));

MouseButton^.EventMask := evMouse;

Insert(MouseButton);

Для очистки события следует обратиться к методу *ClearEvent*, который устанавливает в поле *Event.What* значение *evNothing*, а в поле *Event.InfoPtr* — адрес таблицы виртуальных методов объекта. Таким образом, в поле *Event.InfoPtr* остается «подпись» видимого объекта, который очистил событие. Эта информация используется для организации межэлементного взаимодействия (см. п. 18.7).

18.6.3. Перекрытие *HandleEvent*

После того как Вы определили команду и установили элемент управления, который генерирует ее (например, элемент меню или кнопка диалогового окна), Вам нужно научить Ваш видимый элемент реагировать на возникновение этой команды.

Каждый видимый элемент наследует обработчик событий — метод *HandleEvent*, который уже знает, как реагировать на большую часть ввода пользователя. Если Вы хотите, чтобы видимый элемент делал что-то специфическое для Вашей программы, Вам необходимо перекрыть *HandleEvent* и научить новый обработчик событий двум вещам — как откликаться на определенные Вами команды и как реагировать на события от мыши и клавиатуры нужным Вам образом.

Метод *HandleEvent* определяет поведение видимого элемента. Два видимых элемента с идентичными методами *HandleEvent* будут одинаково откликаться на события. Когда Вы порождаете новый тип видимого элемента, Вы обычно хотите, чтобы его поведение более или менее соответствовало поведению его предка с некоторыми изменениями. Наиболее простой способ достичь этого — вызвать *HandleEvent* предка в методе *HandleEvent* нового объекта.

Общий вид *HandleEvent* наследника:

```
procedure NewDescendant.HandleEvent(var Event: TEvent);
```

```
begin
```

```
  {Код, изменяющий или ограничивающий унаследованное поведение}
```

```
  Inherited HandleEvent(Event);
```

```
  {Код, выполняющий дополнительные функции}
```

```
end;
```

Другими словами, если Вы хотите, чтобы новый объект обрабатывал события не так, как это делал его предок, Вы должны перехватить определенные события до передачи события в метод *HandleEvent* предка. Если Вы хотите, чтобы Ваш новый объект вел себя также, как его предок,

но с дополнительными функциями, Вы должны добавить код **после** вызова процедуры *HandleEvent* предка.

18.6.4. Перекрытие *GetEvent*

Единственным источником событий является метод *TView.GetEvent*. Только этот метод обращается к внешним источникам информации — клавиатуре и мыши. Если в Вашей программе используются другие устройства ввода информации (например, джойстик или коммуникационный канал), Вы должны перекрыть этот метод и научить его работе с нестандартными устройствами.

Проще всего перекрыть метод можно при объявлении нового типа Вашей программы, например:

```
Uses App, ...;
type
  MyProgram = object (TApplication)
    Procedure GetEvent (var Event: TEvent); Virtual;
    .....
  end;
Procedure MyProgram.GetEvent (var Event: TEvent);
begin
  TApplication.GetEvent (Event);
  if Event.What = evNothing then
    begin
      {Обращение к нестандартным источникам информации}
    end
end;
```

Поскольку *MyProgram* в конечном счете наследует *GetEvent* от *TView*, все видимые элементы Вашей программы будут пользоваться новым источником информации.

Преимущества централизованного сбора событий очевидны. Перекрытие единственный метод *GetEvent*, Вы можете заставить программу реагировать на внешнюю информацию любым удобным для Вас способом. Например, можно перехватывать заранее обусловленные командные клавиши и развертывать их в целую серию событий. Таким способом легко создавать различного рода макрокоманды.

18.6.5. Неиспользованное время

Поскольку программа, работающая в среде Turbo Vision, рассчитана на диалог с пользователем, в ней всегда найдутся такие промежутки времени, в течение которых она просто пассивно ожидает действий пользователя. Turbo Vision предоставляет Вам удобное средство, позволяющее в этот момент загрузить процессор не слишком долгой, но нужной для Вас работой.

Стандартный метод *TView.GetEvent* построен таким образом, что если нет никаких событий, он обращается к псевдоабстрактному методу *TView.Idle*. По умолчанию *TView.Idle* ничего не делает, он просто воз—

вращает управление методу *GetEvent*, заставляя его непрерывно сканировать клавиатуру и мышь. Вы можете перекрыть *TView.Idle*, чтобы выполнить нужные действия.

В следующем примере каждые 5 сек в правый верхний угол экрана выводится системное время. Для упрощения программы вывод осуществляется стандартными средствами Турбо Паскаля.

```
Uses DOS,CRT,App;
type
  TProg = object (TApplication)
    Procedure Idle; Virtual;
  end;
Procedure TProg.Idle;
const
  old: Byte = 0; {Старое значение секунд}
  dt = 5; {Шаг вывода}
var
  ho,mi,se,s100: Word;
Function TimeStr(k: Word): String;
var
  s: String [2];
begin
  str(k,s);
  if k < 10 then
    s := '0'+s;
  TimeStr := s
end; {TimeStr}
begin {TProg.Idle}
  GetTime(ho,mi,se,s100);
  if (se mod dt = 0) and (old <> se) then
    begin
      Old := Se;
      TextColor(Black);
      TextBackGround(White);
      GotoXY(72,1);
      Write(TimeStr(ho)+':'+TimeStr(mi)+':'+TimeStr(se))
    end
end; {TProg.Idle}
var
  Prog:TProg;
begin
  Prog.Init;
  Prog.Run;
  Prog.Done
end.
```

Разумеется, не следует поручать методу *TView.Idle* слишком сложную работу, иначе пользователь Вашей программы будет безуспешно нажимать

на клавиши, пытаясь вернуть к жизни «зависшую» программу. Предполагается, что рабочий цикл метода не будет превышать нескольких сотен миллисекунд. Если все-таки Вы хотите выполнить достаточно длинную фоновую задачу, попытайтесь разбить ее на серию мелких шагов.

18.6.6. Ненужные события

Некоторые события могут оказаться ненужными в данном контексте программы. Например, пользователь может нажать командную клавишу, временно запрещенную для использования, или отметить мышью поле вне текущего диалогового окна. Ненужные события — это события, обработка которых не предусмотрена в данном модальном элементе или в любом из его подэлементов. Такие события возвращаются модальному элементу, который в этом случае вызывает свой виртуальный метод *EventError*. Этот метод вызывает метод *EventError* своего владельца и так происходит до тех пор, пока не будет вызван метод *TApplication.EventError*. По умолчанию метод *TApplication.EventError* просто ничего не делает.

Вы можете перекрыть метод *EventError* любого видимого элемента (или программы), чтобы, например, сообщить пользователю о его ошибке и/или дать справку о возможностях программы в данный момент. Кроме того, контроль за ненужными событиями может быть полезен на этапе отладки программы.

18.7. ВЗАИМОДЕЙСТВИЕ ВИДИМЫХ ЭЛЕМЕНТОВ

Иерархия объектов Turbo Vision построена так, чтобы любой объект имел все необходимые для его работы поля и методы. Взаимодействие видимых элементов обычно осуществляется путем создания и использования групп. Если у Вас возникнет необходимость организовать взаимодействие не связанных друг с другом объектов, следует прежде всего тщательно проанализировать программу: возможно Вы не использовали всех средств Turbo Vision или некорректно разделили задачи между двумя видимыми объектами. В большинстве случаев задача может быть решена путем создания нового объекта, объединяющего в себе свойства двух других.

Если программа спроектирована правильно и видимые элементы требуют взаимодействия между собой, можно создать объект-посредник. Типичным примером такого объекта является внутренний буфер *Clipboard* диалоговой среды Турбо Паскаля (опция *Edit*). Для передачи фрагмента текста из одного окна редактора в другое фрагмент помещается в буфер командными клавишами *Shift-Del* или *Ctrl-Ins*, затем вызывается другое окно и фрагмент вставляется в нужное место командой *Shift-Ins*. Вы можете организовать такого рода буфер и в Вашей программе, так как с помощью Turbo Vision Вам доступны все средства диалоговой среды Турбо Паскаля. Преимуществом создания объекта-посредника является то, что с его помощью легко решается проблема взаимодействия сразу нескольких объектов друг с другом. Например, если Вы создаете сложную интегрированную систему, включающую текстовый редактор, систему управления базами данных, электронную таблицу и сервисные средства, промежу-

точный буфер позволит передать данные из текстового редактора в базу данных, или из базы данных в электронную таблицу и т.д.

Другим способом организации взаимодействия элементов является использование событий—сообщений. Эти события создаются с помощью глобальной функции *Message*. Функция *Message* описана в интерфейсной части модуля *Views* следующим образом:

```
Function Message (Receiver: Pview; What, Command: Word;  
                  InfoPtr: Pointer): Pointer;
```

Первым параметром указывается ссылка на экземпляр объекта, которому адресуется сообщение. Остальные параметры используются для создания записи *TEvent*. Функция создает запись события и, если это возможно, вызывает метод *Receiver^.HandleEvent* для обработки этого события. Если адресат не существует или при обращении к функции указан параметр *Receiver = NIL*, функция возвращает *NIL* — это означает, что событие не было обработано. Если событие успешно обработано (*Receiver^.HandleEvent* возвращает событие с полем *What = evNothing*), функция вернет указатель *Event.InfoPtr*.

Как уже говорилось в п.18.6.2, стандартный метод *ClearEvent* очищает событие, устанавливая *Event.What = evNothing* и *Event.InfoPtr = @Self*. Таким образом, объект, обработавший и очистивший событие, оставляет в *Event.InfoPtr* указатель на свою таблицу виртуальных методов. Этот указатель позволяет полностью идентифицировать объект—получатель сообщения и организовать связь с ним.

Рассмотрим пример. В диалоговой среде Турбо Паскаля используется окно отладки, с которым Вы, очевидно, хорошо знакомы. Если программист потребовал открыть это окно, среда должна проверить, открывалось ли оно ранее: если да, то окно просто переносится наверх (в смысле Z-упорядочения), если нет, создается вновь. Для реализации проверки среда дает сообщение

```
AreYouThere := Message(DeskTop, evBroadcast,  
                       cmFindWatchWindow, NIL);
```

В методе *HandleEvent* окна отладки есть проверка на команду *cmFindWatchWindow*:

```
if (Event.What = evBroadcast) and  
    (Event.Command = cmFindWatchWindow) then  
    ClearEvent(Event);
```

Если окно существует, оно очистит событие и оставит в нем сзую «подпись», поэтому сразу за передачей сообщения реализуется такая проверка:

```
if AreYouThere = NIL then  
    CreateWatchWindow      {Создать новое окно}  
else  
    AreYouThere^.Select; {Поместить существующее окно наверх}
```

Поскольку окно отладки — это единственный объект, который знает, как реагировать на команду *cmFindWatchWindow*, в диалоговой среде Турбо Паскаля может использоваться только одно окно этого типа.

Таким же способом компилятор Турбо Паскаля определяет окно редактора, из которого следует брать текст компилируемой программы. Это окно — всегда верхнее на экране, поэтому компилятор посылает сообщение с командой, на которую откликаются только окна редактора. Так как событие — сообщение передается в Z-порядке, первое откликнувшееся окно, т.е. окно, очистившее событие, и будет самым верхним.

Видимый элемент может послать сообщение, установив в поле *Event.InfoPtr* указатель на собственную таблицу виртуальных методов. Это даст возможность получателю сообщения при необходимости обратиться к методам объекта — отправителя.

Организовать взаимодействие можно также с помощью прямого вызова обработчика событий нужного объекта. Для этого экземпляр объекта — адресата (переменная типа объект или указатель на него) должен быть инициализирован предыдущим обращением к конструктору и программа должна «видеть» его, т.е. он должен быть глобальным по отношению к инициатору взаимодействия.

18.8. КОНТЕКСТНАЯ ПОМОЩЬ

В Turbo Vision предусмотрены средства, облегчающие создание контекстно-зависимой справочной службы. С этой целью каждый видимый элемент имеет специальное шестнадцатирядное поле *TView.HelpCtx*, содержимым которого Вы можете распоряжаться по своему усмотрению. Обычно каждому видимому элементу присваивается свой код (целое число в диапазоне от 0 до 65535), устанавливаемый в поле *HelpCtx*. В этом случае при нажатии на заранее обусловленную командную клавишу, открывающую доступ к справочной службе (обычно это клавиша *F1*), программа может получить текущий контекст (прямым чтением поля *HelpCtx* или с помощью метода *GetHelpCtx*) и передать его в качестве параметра вызова справочной службы.

Где обрабатывать событие, связанное с нажатием клавиши вызова справочной службы? Идеальным местом для этого является источник всех событий — метод *GetEvent*. Этот метод связан с любым видимым элементом, в том числе и с терминальным видимым объектом, и поэтому без труда может получить текущий контекст.

В следующем примере на экране создается диалоговое окно с двумя кнопками. Клавиша *F1* используется для доступа к справочной службе. Если активна (выбрана) левая кнопка, нажатие на *F1* даст сообщение «Левая кнопка», если правая — «Правая кнопка». Если на экране нет диалогового окна (оно вызывается клавишей *F2*), появится сообщение «Нет окна».

Uses CRT, App, Dialogs, Objects, Drivers, Views, Menus;

type

 PProg = ^TProg;

 TProg = object (TApplication)

```

Procedure HandleEvent(var Event: Tevent); Virtual;
Procedure GetEvent(var Event: Tevent); Virtual;
Procedure InitStatusLine; Virtual;
end;
Procedure TProg.HandleEvent(var Event: TEvent);
Procedure DialogInit;
var
  R: TRect;
  Dia: PDialog;
  B1,B2: PButton;
  c: Word;
begin
  ClearEvent(Event);
  R.Assign(20,9,60,17);
  Dia := New(PDialog, Init(R, ''));
  R.Assign(3,4,19,6);
  B1 := New(PButton, Init(R, 'Левая', 0, 0));
  B1^.HelpCtx := 1;
  Dia^.Insert(B1);
  R.Assign(20,4,35,6);
  B2 := New(PButton, Init(R, 'Правая', 0, 0));
  B2^.HelpCtx := 2;
  Dia^.Insert(B2);
  c := ExecView(Dia)
end; {DialogInit}
begin{TProg.HandleEvent}
  TApplication.HandleEvent(Event);
  if (Event.What = evCommand)
    (Event.Command = cmMenu) then
    DialogInit
end; {TProg.HandleEvent}
Procedure TProg.InitStatusLine;
var
  R: TRect;
begin
  GetExtent(R);
  R.A.Y := pred(R.B.Y);
  StatusLine := New(PStatusLine, Init(R,
  NewStatusDef(0,0,      {Начальная строка статуса}
  NewStatusKey('~Alt-X~ Выход', kbAltX, cmQuit,
  NewStatusKey('~F1~ Справка', kbF1, cmHelp,
  NewStatusKey('~F2~ Окно диалога', kbF2, cmMenu, NIL))),
  NewStatusDef(1,2,      {Строка статуса с диалоговым окном}
  NewStatusKey('~Esc~ Выход', kbEsc, cmCancel,
  NewStatusKey('~F1~ Справка', kbF1, cmHelp, NIL)),
  NIL)))));
end; {TProg.InitStatusLine}

```

```

Procedure TProg.GetEvent(var Event: TEvent);
const
  txt: array [0..2] of String =
    ('Нет окна', 'Левая кнопка', 'Правая кнопка');
begin
  TApplication.GetEvent(Event);
  if Event.Command = cmHelp then
    begin
      GotoXY(60,1);
      TextColor(Black+Blink);
      TextBackGround(White);
      Write(Txt[GetHelpCtx]);
      Delay(2000);
      GotoXY(60,1);
      Write('          ')
    end
end; {TProg.GetEvent}
var
  Prog: TProg;
begin
  Prog.Init;
  Prog.Run;
  Prog.Done
end.

```

Для упрощения программы справочное сообщение выводится стандартными средствами Турбо Паскаля. Сообщение появляется в верхнем правом углу экрана и через 2 сек стирается, в течение этого промежутка времени доступ к клавиатуре и мыши блокируется.

Контекст видимого элемента может управлять содержимым строки статуса. С этой целью в методе *InitStatusLine* используется два или более вызова *NewStatusDef* со своим диапазоном контекста (см. предыдущий пример).

В состав демонстрационных модулей Turbo Vision включен файл *HelpFile.pas* и компилятор *TVHC.pas*, существенно упрощающие процесс создания и использования контекстно-чувствительной помощи.

Средства модуля *HelpFile* позволяют по заданному контексту найти в особом файле справочной службы нужную справку и вывести ее на экран. Текст справки помещается в окно со скроллером, границы которого могут изменяться. Справочная служба контролирует размер окна и форматирует текст так, чтобы строки не выходили за границы окна. В тексте справки могут использоваться перекрестные ссылки, представляющие собой выделенные цветом фрагменты текста. Справочная служба обеспечивает доступ к этим ссылкам так, как это реализовано в среде Турбо Паскаля: ссылку можно выбрать клавишей *Tab* или отметкой мышью, после чего нажатие на *Enter* приведет к раскрытию нового справочного окна с соответствующей справкой.

Для использования этих средств необходимо предварительно создать файл справочной службы, содержащий текстовые данные и средства управления ими. Файл справочной службы создается программой *TVHC.PAS* из обычного текстового файла, в котором используются следующие соглашения:

- если очередная строка начинается символами `.topic`, эта строка определяет заголовок темы;
- если в тексте встречается фрагмент, обрамленный фигурными скобками, этот фрагмент определяет перекрестную ссылку.

Заголовок темы содержит идентифицирующее справку уникальное кодовое слово, и связанный с ним контекст. Например:

```
.topic Viewer=2
```

Здесь *Viewer* — кодовое слово; 2 — контекст справочной службы.

Все остальные строки до очередного заголовка темы составляют справочное сообщение и будут выводиться в окне справочной службы. Если очередная строка справки начинается пробелом, эта строка не форматируется и может отсекается границами окна, в противном случае строка выводится с учетом текущих границ окна (если очередное слово не умещается в окне, остаток текстовой строки переносится на следующую строку окна). Например:

```
.topic FileOpen=3  
File Open  
-----
```

Эта опция меню используется для загрузки файла

Здесь строки

```
File Open  
-----
```

начинаются пробелом и поэтому не форматируются, т.е. сохраняют свой вид независимо от границ окна, в то время как при выводе строки

Эта опция меню используется для загрузки файла

будут контролироваться границы окна (строка не начинается пробелом) и, если очередное слово выйдет за его правую границу, остаток строки вместе с неуместившимся словом будет выведен в следующей строке окна.

Любой фрагмент строки справочного сообщения, обрамленный фигурными скобками, рассматривается как перекрестная ссылка. Перекрестная ссылка может содержать кодовое слово заголовка нужной справки или произвольный текст, за которым указывается двоеточие и кодовое слово. Например:

С помощью опции {FileOpen} можно загрузить файл.

или

Окно обеспечивает {просмотр файла: Viewer} в режиме скроллера.

В первом случае будет выведена строка

С помощью опции **FileOpen** можно загрузить файл.

причем слово *FileOpen* будет выделено цветом и может использоваться как перекрестная ссылка на справку, заголовок темы которой имеет вид

```
.topic FileOpen = ...
```

Во втором случае в окне выводится строка

Окно обеспечивает **просмотр файла** в режиме скроллера.

Слова *просмотр файла* выделяются цветом и используются как перекрестная ссылка на справку с заголовком

```
.topic Viewer = ...
```

В заголовке темы можно указывать несколько кодовых слов, например:

```
.topic FileOpen=3, OpenFile=103, FFileOpen
```

Доступ к такой справке возможен для контекстов

```
const
  chFileOpen = 3;
  chOpenFile = 103;
  chFFileOpen = 104;
```

Обратите внимание: за кодовым словом *FFileOpen* не указывается контекст, в этом случае считается, что связанный с ним контекст есть предыдущий контекст, увеличенный на 1, т.е.

```
chFFileOpen = 103 + 1 = 104
```

Файл *DEMOHELP.TXT* содержит пример исходного текстового файла, подготовленного с учетом описанных требований для преобразования программой *TVHC.PAS* в файл справочной службы.

Подготовка справочного файла и его использование осуществляется в следующей последовательности.

1) Подготавливается текстовый файл *NAMETEXT*, содержащий заголовки тем, справки и перекрестные ссылки.

2) Вызывается программа *TVHC.EXE* (исходный текст этой программы содержится в файле *\BP\EXAMPLES\DOS\TVDEMOS\TVHC.PAS*) для создания файла справочной службы *NAMEHELP* и вспомогательного модуля *NAMEPAS*. Обращение к программе описано ниже.

3) В программе, использующей средства модуля *HelpFile*:

- а) вызывается процедура *RegisterHelpFile* для регистрации объектов модуля *HelpFile* в потоке;
- б) открывается поток, связанный с созданным файлом *NAMEHELP*;
- в) создается экземпляр объекта *THelpFile* и ему передается поток и нужный контекст;
- г) иницируется работа созданного экземпляра;
- д) ликвидируется экземпляр объекта *THelpFile*.

Действия 3,б, ..., 3,д осуществляются каждый раз, когда необходимо выдать ту или иную справку.

Для преобразования текстового файла во внутренний формат, используемый средствами модуля *HelpFile*, предназначена программа, исходный текст которой поставляется в файле *TVHC.PAS*. Перед использованием этой программы ее необходимо оттранслировать в *EXE*-файл. Формат вызова программы *TVHC.EXE*:

```
TVHC NAMETEXT [NAMEHELP [NAMEPAS]]
```

(в квадратных скобках указаны необязательные параметры). Здесь *NAMETEXT* — имя исходного текстового файла, *NAMEHELP* — имя выходного файла справочной службы, *NAMEPAS* — имя выходного файла, содержащего текст модуля с контекстами в виде констант *chXXXX*. Если имена выходных файлов опущены, будут созданы файлы с именем исходного файла и расширением *HLP* для файла справочной службы, *PAS* для текста модуля.

Текст файл *NAMEPAS* имеет следующий вид:

```
unit namepas;  
interface  
const  
    chTopic1 = n1;  
    chTopic2 = n2;  
    .....  
    chTopicN = nN;  
implementation  
end.
```

Поскольку этот файл создается программой *TVHC.EXE* автоматически, будьте внимательны при обращении к ней. Не рекомендуется опускать имена *NAMEHELP* и *NAMEPAS*: если имя исходного текстового файла совпадает с именем программы или любого другого *PAS*-файла, старый файл будет стерт и на его месте будет создан файл с текстом модуля!

Для иллюстрации техники использования средств модуля *HelpFile* рассмотрим следующий пример. Пусть текст справочной службы помещен в файл *HelpTest.txt* и имеет следующий вид:

```
.topic NoContext=0  
Добро пожаловать  
в справочную службу системы Turbo Vision!  
  
В текстовом файле для справочной службы Вы должны подгото-  
вить {заголовки тем: topic} и {перекрестные ссылки:  
CrossRef}. Весь текст от одного заголовка до другого  
представляет собой текст справки и выводится в окне спра-  
вочной службы. При подготовке текста учитываются следую-  
щие соглашения:  
если очередная строка начинается символом пробел, эта  
строка не будет форматироваться с учетом границ окна;
```

во всех остальных случаях выводимый в окне текст формируется с учетом границ окна: слово, не уместившееся на строке, переносится на следующую строку экрана. Для преобразования текстового файла в файл справочной службы вызывается программа *TVHC.EXE*:

```
TVHC NAMETXT [NAMEHELP [NAMEPAS]]
```

Здесь *NAMETXT* - имя исходного текстового файла; *NAMEHELP* - имя выходного файла справочной службы; *NAMEPAS* - имя выходного файла, содержащего текст *PAS*-модуля с определением всех контекстов в виде констант *chXXXX*.

Имя файла *NAMEHELP* можно не указывать - в этом случае выходной файл будет иметь имя входного файла и расширение *HLP*. Если не указан файл *HELPPAS*, будет создан файл с именем исходного и расширением *PAS*.

По умолчанию программа использует расширение *TXT* для входного файла, *HLP* для выходного файла справочной службы и *PAS* - для файла констант.

```
.topic Topic=1
```

Заголовок темы имеет следующий формат:

```
.topic Name[=N][, Name1[=N2][...]]
```

Здесь *Name* - имя темы (может состоять только из латинских букв и цифр; разница в высоте букв игнорируется); *N* - контекст справочной службы, при котором выдается данная справка.

В квадратных скобках показаны необязательные параметры. Если опущен контекст *N*, программа присваивает соответствующей перекрестной ссылке контекст *NPrev* + 1, где *NPrev* - предыдущий определенный в программе контекст.

```
.topic CrossRef=2
```

В произвольное место текста справки можно вставить так называемую перекрестную ссылку:

```
(text[:title])
```

Здесь *()* - фигурные скобки; *text* - произвольный текст или заголовок темы; *title* - заголовок темы; этот параметр вместе с двоеточием опускается, если *text* - заголовок темы.

Если Вы подготовите такой файл, то после вызова

```
tvhc helptest
```

будут созданы два файла: файл справочной службы *helptest.hlp* и файл модуля с определениями констант *helptest.pas*. Содержимое этого последнего файла будет таким:

```

unit helptest;
interface
const
    hcCrossRef          = 2;
    hcNoContext         = 0;
    hctopic             = 1;
implementation
end.

```

Следующая программа будет выдавать контекстно-зависимые справочные сообщения из файла *helptest.hlp* при каждом нажатии на *F1*.

Uses App, Menus, Drivers, Views, Objects, HelpFile, Dialogs;

const

cmChangeCtx = 1000;

type

PCtxView = ^TCtxView;

TCtxView = **object** (TView)

Constructor Init;

Procedure Draw; **Virtual**;

end;

MyApp = **object** (TApplication)

 Ctx: PCtxView;

Constructor Init;

Procedure InitStatusLine; **Virtual**;

Procedure HandleEvent(**var** Event: TEvent); **Virtual**;

end;

PMyHelpWindow = ^MyHelpWindow;

MyHelpWindow = **object** (THelpWindow)

Function GetPalette: PPalette; **Virtual**;

end;

procedure MyApp.InitStatusLine;

var

 R: TRect;

begin

 GetExtent(R);

 R.A.Y := pred(R.B.Y);

 StatusLine := New(PStatusLine, Init(R,

 NewStatusDef(0, \$FFFF,

 NewStatusKey('~Alt-X~ Выход', kbAltX, cmQuit,

 NewStatusKey('~F1~ Помощь', kbF1, cmHelp,

 NewStatusKey('~F2~ Изменить контекст', kbF2,

 cmChangeCtx,

 NewStatusKey('~F5~ Распахнуть окно', kbF5,

 cmZoom,

 NIL))));

 NIL)))

end;

```

Constructor MyApp.Init;
begin
    TApplication.Init;
    Ctx := New(PCtxView, Init);
    Insert(Ctx);
    RegisterHelpFile
end;
Procedure MyApp.HandleEvent;
var
    HF: PHelpFile;
    HS: PDosStream;
    HW: PMyHelpWindow;
const
    HelpOpen: Boolean = False;
Procedure DoHelp;
    {Обеспечивает доступ к контекстно-зависимой справочной службе}
var
    C: Word;
begin
    ClearEvent(Event);
    { Открываем DOS-поток: }
    HS := New(PDosStream, Init('HELPTTEST.HLP', stOpenRead));
    { Создаем и иницилируем экземпляр объекта THelpFile: }
    HF := New(PHelpFile, Init(HS));
    HelpOpen := HS^.Status = stOk;
    if HelpOpen then
        begin
            {Создаем окно справочной службы и связываем его с потоком H.S. и
            текущим контекстом:}
            HW := New(PMyHelpWindow, Init(HF, GetHelpCtx));
            if ValidView(HW) <> NIL then
                begin
                    C := ExecView(HW);           {Выдаем справку}
                    Dispose(HW)                 {Ликвидируем окно}
                end;
                HelpOpen := False
            end
        else
            Dispose(HF, Done)
        end;
begin
    TApplication.HandleEvent(Event);
    case Event.Command of
        cmHelp: if not HelpOpen then
            DoHelp;                               {Выдаем справку}
        cmChangeCtx:      {Изменяем контекст по клавише F2}
            begin
                if HelpCtx = 3 then

```

```

        HelpCtx := 0
    else
        inc(HelpCtx);
        Ctx^.Draw;
        ClearEvent(Event)
    end
end
end;
Function MyHelpWindow.GetPalette;
const
    P = #16#17#18#19#20#47#21#13;
    C: String [8] = P;
begin
    GetPalette := @C
end;
Constructor TCtxView.Init;
var
    R: TRect;
begin
    R.Assign(67,0,80,1);
    TView.Init(R);
end;
Procedure TCtxView.Draw;
var
    S: String;
    B: TDrawBuffer;
    C: Byte;
begin
    Str(Application^.HelpCtx,S);
    S := 'Контекст = '+S;
    C := GetColor(2);
    MoveChar(B, ' ', C, Size.X);
    MoveStr(B, S, C);
    WriteLine(0,0,Size.X,1,B)
end;
var
    P: MyApp;
begin
    P.Init;
    P.Run;
    P.Done
end.

```

В программе предусмотрена смена текущего контекста с помощью клавиши *F2*. Разумеется, в реальной программе смена контекста будет, судя по всему, происходить иначе: обычно в текст конструктора *Init* иди-
дического элемента вставляется оператор

HelpCtx := NNN;

где NNN — нужный контекст.

Для визуализации текущего контекста в программе используется простой объект *TCtxView*, с помощью которого в верхнем правом углу экрана выводится строка

Контекст = N

N — текущий контекст.

КОЛЛЕКЦИИ

Одним из недостатков языка Паскаль (и Турбо Паскаль) является не — возможность создания и использования в программе массивов с переменной размерностью — так называемых динамических массивов. Этот недостаток особенно ощутимо сказывается в диалоговых программах, работа которых в существенной своей части определяется действиями пользователя. Если программист заранее не знает, какие именно требования к используемым в программе массивам предъявит пользователь, он обычно резервирует для них максимально возможные объемы памяти или размещает массивы в куче. И тот и другой способы нельзя считать вполне удовлетворительными: в первом случае возникают неестественные ограничения на предельно возможные размеры массивов или выделенная под их размещение память расходуется нерационально, во втором случае приходится прибегать к дополнительным ухищрениям, чтобы организовать индексированный доступ к динамической памяти.

Разработчики Turbo Vision решили проблему кардинально, создав механизм коллекций. Хотя экземпляры объектов можно объединять в массивы, как и «обычные» переменные Паскаля, Вы вряд ли захотите использовать массивы для их хранения: коллекции не только снимают проблемы фиксированных границ, но имеют еще и целый ряд новых возможностей, которыми массивы не обладают.

Изучению коллекций посвящается эта глава.

19.1. ЭЛЕМЕНТЫ КОЛЛЕКЦИЙ

Коллекции предназначены, строго говоря, для тех же целей, что и массивы языка Турбо Паскаль: они позволяют создать набор из произвольного количества элементов и организуют индексный способ доступа к этим элементам. В отличие от массивов коллекции обладают двумя новыми свойствами. Во — первых, их размер может динамически меняться в ходе работы программы, фактически ограничиваясь лишь доступной памятью. Во — вторых, в коллекции могут храниться элементы разных типов. Последнее свойство называется полиморфизмом коллекций.

Технически коллекции представляют собой массивы нетипизированных указателей на размещенные в динамической памяти элементы коллекций. Эти массивы размещаются в куче — отсюда возможность динамического изменения размеров коллекций; с другой стороны, входящие в эти списки указатели позволяют ссылаться на произвольные элементы, отсюда полиморфизм.

Полиморфизм коллекций — это очень мощное средство, которым следует пользоваться с осторожностью, поскольку фактически коллекция хранит лишь указатели на элементы. Компилятор не может проверить правильность доступа к элементам; Вы можете поместить в коллекцию

один объект, а взять его из коллекции как объект другого типа и компилятор не сможет предупредить Вас об этом.

19.2. СОЗДАНИЕ КОЛЛЕКЦИЙ

Для создания коллекции следует прежде всего определить тип тех данных, которые будут в ней храниться. Например, Вам необходимо создать электронный каталог Вашей личной библиотеки. В этом случае для каждой указанной в каталоге книги можно выделить пять полей данных: автор, название, издательство, год издания и количество страниц. Создадим новый объект *TBook* следующего вида:

Uses Objects;

type

PBook = ^TBook;

TBook = **object** (TObject)

Autor: PString; {Автор}

Title: PString; {Название}

PubHouse: PString; {Издательство}

Year: Word; {Год издания}

Pages: Word; {Количество страниц}

Constructor Init (A, T, PH: **String**; Y, P: Word);

Destructor Done; **Virtual**;

end;

Мы включили в объект поля и два основных метода работы с ними: конструктор *Init* используется для размещения в динамической памяти очередной записи каталога, деструктор *Done* удаляет записи из кучи. Заметим, что в объекте задаются не сами текстовые строки, а указатели на них (тип *PString* в Turbo Vision описан как указатель на тип *String*). Такое размещение данных как правило дает значительную экономию памяти. Например:

Constructor TBook.Init (A, T, PH: **String**; Y, P: Word);

begin

Autor := NewStr(A);

Title := NewStr(T);

PubHouse := NewStr(PH);

Year := Y;

Pages := P

end; {TBook.Init}

Используемые в конструкторе функции *NewStr* размещают в динамической памяти текстовую строку, выделяя под нее минимально необходимую память, что значительно выгоднее по сравнению с типичным описанием текстовых полей вида

type

TBook = **object** (TObject)

Autor, Title, PubHouse: **String**;

```
.....
end;
```

Для освобождения динамической памяти в деструкторе *Done* используется процедура *DisposeStr*:

```
Destructor TBook.Done;
begin
  DisposeStr(Autor);
  DisposeStr(Title);
  DisposeStr(PubHouse)
end; {TBook.Init}
```

После того как тем или иным способом определены типы данных, создание коллекции не вызывает проблем. Например:

```
var
  BookList: PCollection;
begin
  BookList := New(PCollection, Init(50,10));
  with BookList^ do
    begin
      Insert(New(PBook, Init('Джордейн Р.',
        'Справочник программиста персональных компьютеров'+
        ' типа IBM PC, XT и AT', 'Финансы и статистика',
        1991,544)));
      Insert(New(PBook, Init('Шелдон',
        'Язык Си для профессионалов', 'И.В.К.-СОФТ', 1991,383)));
      Insert(New(PBook, Init('Скэнлон Л.',
        'Персональные ЭВМ IBM PC и XT. '+
        'Программирование на языке ассемблера',
        'Радио и связь', 1991,336)));
      Insert(New(PBook,
        Init('Йенсен К., Вирт Н.',
        'Паскаль. Руководство для пользователя '+
        'и описание языка', 'Финансы и статистика', 1982,151))),
    end;
  .....
  Dispose(BookList, Done);
end;
```

Для создания коллекции мы обратились к методу *TCollection.Init*, указав ему начальную длину коллекции (50 элементов) и шаг наращивания (10 элементов). Руководствуясь этими указаниями, Turbo Vision зарезервирует в динамической памяти место для размещения 50 указателей. Если в ходе наполнения коллекции ее длина превысит начальную, Turbo Vision будет наращивать коллекцию порциями, каждая из которых достаточна для размещения 10 указателей.

Смысл параметров, передаваемых методу *TCollection.Init*, станет понятнее, если рассмотреть механизм создания и обновления коллекции.

Вначале в куче резервируется участок памяти, достаточный для размещения массива из $N0$ указателей ($N0$ — начальный размер коллекции). Если в ходе наполнения коллекции ее длина превысит $N0$ элементов, резервируется новый участок памяти, достаточный для размещения массива из $N0 + DN$ указателей (DN — шаг наращивания коллекции), затем старый массив переносится на новое место, а память, выделенная под его размещение, возвращается в кучу. Таким образом, чем больше начальная длина коллекции и шаг ее наращивания, тем меньше суммарные потери времени на расширение коллекции, но и тем больше могут стать потери памяти, если реальная длина коллекции окажется значительно меньше $N0 + k \cdot DN$ ($k = 0, 1, 2, \dots$).

Операторы *Insert* размещают в динамической памяти элементы коллекции. В реальной программе наполнение коллекции будет, судя по всему, осуществляться каким-то иным способом, чем простое программирование обращений к методу *Insert* (см., например, программу *Notebook* из гл.15). Для нас сейчас важно другое: мы нигде не говорили коллекции, какого типа объекты она будет хранить; для обеспечения нужных действий по размещению в памяти очередного элемента мы просто обращаемся к соответствующему методу *Init*, а уж он делает остальное — ведь он «знает» как это следует сделать.

Отметим, что обращение

```
Dispose(BookList, Done);
```

вызывает автоматическое обращение к методу *TBook.Done* перед уничтожением каждого элемента коллекции, после чего уничтожается экземпляр *TCollection*. Это стало возможным потому, что объект *TBook* объявлен нами как потомок от *TObject*. Если бы мы его объявили независимым объектом

```
type
  TBook = object :
    .....
  end;
```

мы должны были бы сами позаботиться об освобождении кучи, а обращение

```
Dispose(BookList, Done);
```

привело бы к «зависанию» программы.

19.3. ДОСТУП К ЭЛЕМЕНТАМ КОЛЛЕКЦИЙ

Итак, оператором

```
BookList := New(PCollection, Init(50,10));
```

мы объявили о создании коллекции, а операторами

```
Insert(New(PBook, Init(...)))
```

наполнили эту коллекцию нужными элементами. Как осуществить доступ к элементам коллекции? Для этого можно использовать несколько способов.

Во-первых, к любому элементу коллекции можно обратиться по его порядковому номеру (индексу). В отличие от массивов Турбо Паскаля, индексы которых могут иметь произвольные границы, коллекции индексируются целыми числами в диапазоне от 0 до *Count*-1 (*Count* — общее количество элементов в коллекции). Любая коллекция имеет поле *Count*, которое указывает текущую длину коллекции. Чтобы по индексу получить доступ к нужному элементу, используется метод *At*, который возвращает указатель на элемент.

Например, нам требуется вывести на экран содержимое тех записей нашего каталога, которые соответствуют 1991 году издания. Тогда вместо точек в конце программы, показанной на с.455, следует вставить оператор

```
PrintYear(BookList,1991);
```

Кроме того, в раздел описаний программы добавим две новых процедуры:

```
Procedure PrintItem(A,T,PB: String; Y,P: Word);  
{Выводит на экран элемент коллекции}  
begin  
  WriteLn(A);  
  WriteLn(' ',T);  
  WriteLn(' ',PB,' ',Y,' ',P)  
end; {PrintItem}  
Procedure PrintYear(BookList: PCollection; Y: Word);  
{Выводит на экран издания нужного года выпуска}  
var  
  Book: PBook;  
  k: Integer;  
begin  
  WriteLn;  
  for k := 0 to pred(BookList^.Count) do  
    begin  
      Book := BookList^.At(k);  
      with Book^ do if Year = Y then  
        PrintItem(Autor^,Title^,PubHouse^,Year,Pages)  
    end  
end; {PrintYear}
```

В процедуре *PrintYear* организуется счетный цикл от 0 до *pred* (*TCollection.Count*). С помощью оператора

```
Book := BookList^.At(k);
```

в переменную *Book* помещается указатель на элемент коллекции с индексом *k*. Именно здесь обеспечивается полиморфизм коллекций: метод *At* возвращает нетипизированный указатель, который Вы можете интерпретировать нужным Вам образом. Однако здесь же таится источник трудно обнаруживаемых ошибок: в левую часть оператора присваивания

можно поместить указатель любого типа и тип этого указателя может не соответствовать типу k -го элемента коллекции.

Отметим, что обращение к методу *At* с индексом, выходящим за границы коллекции, активизирует вызов метода *TCollection.Error*, который по умолчанию аварийно завершает исполнение программы (подробнее см. п.17.6).

Помимо использования метода *At* коллекции обеспечивают доступ к трем итерационным методам, которые могут оказаться весьма удобными. Метод *ForEach* осуществляет некоторую заранее заданную операцию сразу над всеми элементами коллекции, а методы *FirstThat* и *LastThat* отыскивают в коллекции первый элемент, удовлетворяющий некоторому опять же заранее заданному критерию поиска: *FirstThat* ищет от начала коллекции к ее концу, а *LastThat* — в обратном направлении.

Чтобы воспользоваться методом *ForEach* мы должны сначала создать процедуру без параметров, которая осуществляет нужные действия над всеми элементами коллекции, и передать адрес этой процедуры как параметр обращения к *ForEach*. Например, для того чтобы вывести на экран содержимое всего нашего каталога, мы должны внести в программу следующие изменения. Поместите в раздел описаний следующий текст процедуры *PrintAll*:

```

Procedure PrintAll(Book: PCollection);
  {Вывод всех элементов коллекции}
Procedure PrintBook(P: PBook); far;
begin
  with P do
    PrintItem(Autor^, Title^, PubHouse^, Year, Pages)
end; {PrintBook}
begin {PrintAll}
  WriteLn;
  Book^.ForEach(@PrintBook);
end; {PrintAll}

```

Как видите, эта процедура содержит внутреннюю процедуру *PrintBook*, в которой осуществляется нужное нам действие — вывод очередного элемента каталога на экран. Этот вывод достигается с помощью вызова уже использованной нами ранее процедуры *PrintItem*. Таким образом, описание процедуры *PrintAll* должно следовать после описания *PrintItem*, чтобы этот вызов был синтаксически правильным. Далее, вывод всех элементов коллекции в процедуре *PrintAll* осуществляется оператором

Book[^].ForEach(@PrintBook);

который обращается к методу *TCollection.ForEach*, передавая ему в качестве параметра адрес процедуры *PrintBook*. Чтобы программа успешно выполнила нужные действия, процедура, адрес которой передается методу *ForEach*, должна удовлетворять двум условиям:

- она должна быть рассчитана на дальнюю модель вызова (для этих целей мы указали директиву **far** сразу за заголовком *PrintBook*);

- она должна быть локальной для процедуры, в которой реализуется вызов *ForEach*, именно поэтому мы разместили ее в теле процедуры *PrintAll*.

Осталось в тело главной программы поместить оператор

```
PrintAll(BookList);
```

перед оператором уничтожения коллекции. Если Вы запустите таким образом подготовленную программу на счет, на экран будет выведено:

Джордейн Р.

Справочник программиста персональных компьютеров типа IBM PC,
ХТ и АТ

Финансы и статистика, 1991, 544

Шелдон

Язык Си для профессионалов

И.В.К.-СОФТ, 1991, 383

Скэнлон Л.

Персональные ЭВМ IBM PC и ХТ. Программирование на языке ас-
семблера

Радио и связь, 1991, 336

Йенсен К., Вирт Н.

Паскаль. Руководство для пользователя и описание языка
финансы и статистика, 1982, 151

Все действия по выводу содержимого коллекции реализуются методом *ForEach*, который для собственно вывода каждого очередного элемента обращается к процедуре *PrintBook*.

Точно таким же образом реализуется обращение к методам *FirstThat* и *LastThat*. Например, если нам потребуется отыскать в каталоге запись, содержащую слово «Вирт» в поле *Autor*, можно добавить в программу следующие строки

```
Procedure SearchAutor(BookList: Pcollection; A: String);  
Function FindAutor(P: PBook): Boolean; far;  
begin  
    FindAutor := pos(A, P^.Autor^) <> 0  
end; {FindAutor}  
var  
    Book: PBook;  
begin {SearchAutor}  
    Book := BookList^.FirstThat(@FindAutor);  
    if Book = NIL then  
        WriteLn('Нет автора ',A)  
    else with Book^ do  
        begin  
            WriteLn;  
            PrintItem(Autor^,Title^,PubHouse^,Year,Pages)  
        end  
    end;  
end; {SearchAutor}
```

В тело главной программы следует добавить оператор

```
SearchAutor (BookList, 'Вирт');
```

Собственно поиск элемента коллекции реализуется оператором

```
Book := BookList^.FirstThat (@FindAutor);
```

который для этих целей обращается к методу *TCollection.FirstThat*. В этом методе реализуется последовательный анализ всех элементов коллекции, начиная с самого первого (с индексом 0), причем для анализа используется вызов функции *FindAutor*. Как видим, эта функция нужным образом анализирует очередную запись и возвращает *True*, если условие поиска удовлетворено. Метод *FirstThat* возвращает указатель на элемент коллекции, для которого удовлетворено условие поиска, или *NIL*, если этому условию не отвечает ни один элемент. Таким образом, оператор

```
if Book = NIL then
.....
else
.....
```

проверяет результат поиска и выводит на печать найденный элемент коллекции или сообщение «Нет автора ...», если условие поиска не удовлетворено.

Как и в случае метода *ForEach*, функция, передаваемая методу *FirstThat*, должна транслироваться в расчете на дальнюю модель памяти и должна локализоваться в теле процедуры, в которой осуществляется вызов метода.

Любой элемент коллекции можно удалить или заменить новым. Для удаления используется метод *AtFree*, которому в качестве параметра передается индекс удаляемого элемента. При удалении элемента предполагается, что коллекция содержит указатели на объекты, порожденные от *TObject* и размещенные в куче, поэтому автоматически вызывается метод *TObject.Done*. Индексы всех элементов, размещенных в коллекции после удаляемого элемента, уменьшаются на 1.

С помощью метода *DeleteAll* удаляются все элементы из коллекции но сама коллекция при этом сохраняется, т.е. очищенная коллекция будет иметь *Count* = 0. Для очистки коллекции вызывается *AtFree* для каждого элемента.

Чтобы заменить существующий элемент новым, используется метод *AtPut (Index, Item)*, где *Index* — индекс заменяемого элемента, а *Item* — указатель на новый элемент.

Метод *AtInsert (Index, Item)* вставляет новый элемент в коллекцию в позицию *Index* и увеличивает индексы всех ранее существовавших в коллекции элементов от элемента *Index* до конца коллекции на единицу, т.е. «раздвигает» коллекцию.

19.4. ОТСОРТИРОВАННЫЕ КОЛЛЕКЦИИ

Часто бывает необходимо каким-либо образом упорядочить коллекцию, т.е. расставить ее элементы в определенном порядке. Для этих целей

в Turbo Vision предусмотрен специальный объект *TSortedCollection*. Этот объект порожден от *TCollection* и, следовательно, уже умеет создавать коллекцию, вставлять в нее элементы и удалять их. Единственное, чего он не умеет — это сортировать коллекцию. В *TSortedCollection* есть абстрактный метод *Compare*, который используется для упорядочения элементов и который Вы должны перекрыть, чтобы обеспечить нужную Вам сортировку. Таким образом, чтобы создать отсортированную коллекцию, Вы должны создать объект — потомок от *TSortedCollection* и перекрыть его метод *Compare*.

По умолчанию этот метод получает в качестве параметров указатели на два элемента коллекции и должен вернуть 1, 0 или -1 в зависимости от того, больше, равно или меньше какое-то поле первого элемента по сравнению с этим же полем второго элемента. Поле, по которому сравниваются элементы, называется ключевым.

Например, нам требуется создать отсортированную коллекцию, содержащую каталог библиотеки (см. пример п.19.3), причем в качестве ключевого используется поле *Autor*. Тогда создадим новый объект

type

PSort = ^*Tsort*;

TSort = **object** (*TSortedCollection*)

Function *Compare*(*Key1*, *Key2*: *Pointer*): *Integer*; **Virtual**;

end;

чтобы перекрыть метод *Compare*. Если теперь объявить новый метод *TSort.Compare* следующим образом:

Function *TSort.Compare*(*Key1*, *Key2*: *Pointer*): *Integer*;

var

A: *PSort* absolute *Key1*;

B: *PSort* absolute *Key2*;

begin

if *A^.Autor* < *B^.Autor* **then**

Compare := -1

else if *A^.Autor* = *B^.Autor* **then**

Compare := 0

else

Compare := 1

end; {*TSort.Compare*}

то после объявления

var

BookList: *PSort*;

вместо

var

BookList: *PCollection*;

программа выведет каталог, отсортированный по фамилиям авторов:

Джордейн Р.

Справочник программиста персональных компьютеров типа IBM PC, XT и AT

Финансы и статистика, 1991, 544

Йенсен К., Вирт Н.

Паскаль. Руководство для пользователя и описание языка

Финансы и статистика, 1982, 151

Скэнлон Л.

Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблера

Радио и связь, 1991, 336

Шелдон

Язык Си для профессионалов

И.В.К.-СОФТ, 1991, 383

Ключевое поле определяется методом *TSortedCollection.KeyOf*. Этот метод по заданному в качестве параметра обращению указателю на элемент коллекции возвращает указатель на ключевое поле. По умолчанию метод *KeyOf* возвращает указатель на весь элемент, однако Вы можете перекрыть его новым методом, возвращающим указатель на нужное ключевое поле. Пусть, например, нам требуется отсортировать каталог по году издания книг (поле *Year*). Добавим в описание объекта *TSort* перекрытие метода *KeyOf*:

type

TSort = **object** (TSortedCollection)

.....

Function KeyOf(Item: Pointer): Pointer; **Virtual**;

end;

Опишем новый метод следующим образом:

Function TSort.KeyOf(Item: Pointer): Pointer;

begin

KeyOf := @PBook(Item)^.Year

end;

и изменим описание метода *Compare*:

Function TSort.Compare(Key1, Key2: Pointer): Integer;

var

A: ^Integer **absolute** Key1;

B: ^Integer **absolute** Key2;

begin

if A^ < B^ **then**

Compare := -1

else if A^ = B^ **then**

Compare := 0

else

Compare := 1

```
end; {TSort.Compare}
```

Теперь после запуска программы на экран будет выведено:

Йенсен К., Вирт Н.

Паскаль. Руководство для пользователя и описание языка
финансы и статистика, 1982, 151

Джордейн Р.

Справочник программиста персональных компьютеров типа IBM PC,
XT и AT

финансы и статистика, 1991, 544

Обратите внимание: в отсортированной коллекции теперь хранятся только 2 элемента! Произошло это потому, что по умолчанию *TSortedCollection* игнорирует новую запись, если в коллекции уже существует элемент, ключевое поле которого имеет такое же значение. Таким образом, обычно в отсортированной коллекции содержатся записи с уникальными ключевыми полями.

Можно ли поместить в коллекцию два или больше элементов с одинаковыми полями? Turbo Vision позволяет сделать это: поле *TSortedCollection.Duplicates* по умолчанию содержит *FALSE*, что указывает на уникальность ключевого поля; если перед наполнением коллекции Вы поместите в это поле значение *TRUE*, коллекция не будет контролировать уникальность ключевых полей.

Изменим начало раздела исполняемых операторов главной программы следующим образом:

```
begin
```

```
  BookList := New(PSort, Init(50,10));
```

```
  with BookList^ do
```

```
    begin
```

```
      Duplicates := True; {Отменяем уникальность ключей}
```

```
      .....
```

```
    end;
```

```
    .....
```

```
end.
```

Теперь на экран будет выведено:

Йенсен К., Вирт Н.

Паскаль. Руководство для пользователя и описание языка
финансы и статистика, 1982, 151

Скэнлон Л.

Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблера

Радио и связь, 1991, 336

Шелдон

Язык Си для профессионалов

И.В.К.-СОФТ, 1991, 383

Джордейн Р.

Справочник программиста персональных компьютеров типа IBM PC, XT и AT

Финансы и статистика, 1991, 544

Заметим, что, очередной элемент вставляется перед первым элементом с равным значением ключевого поля. Точно также поиск *First.That* вернет указатель на первый из нескольких элементов с одинаковыми ключевыми полями, а метод *LastThat* — на последний из них.

19.5. КОЛЛЕКЦИИ СТРОК

Для создания и использования коллекции отсортированных строк в Turbo Vision используется объект *TSStringCollection*. Этот объект является прямым потомком от *TSortedCollection* и отличается от него тем, что его метод *Compare* не является абстрактным — по умолчанию он осуществляет обычное для Турбо Паскаля лексикографическое сравнение двух строк. Таким образом, если Вам необходимо отсортировать коллекцию строк по алфавиту (точнее, в соответствии с внутренней кодировкой символов), Вы можете использовать экземпляр объекта *TSortedCollection* без какого-либо перекрытия его методов.

В следующей программе создается словарь слов, входящих в некоторый текстовый файл. По умолчанию используется файл с текстом программы, но Вы можете указать имя любого текстового файла в качестве параметра вызова программы.

Uses Objects;

var

f: file of Char;

Function OpenFile(**var** Name: String): Boolean;

{Возвращает FALSE, если нельзя открыть файл}

begin

if ParamCount = 1 then

Name := ParamStr(1) {Первый параметр в строке вызова
программы должен содержать имя файла,}

else {если это не так, анализируется файл,
содержащий текст программы}

Name := copy(ParamStr(0), 1,
pos('.', ParamStr(0))) + 'PAS';

Assign(f, Name);

{SI-}

Reset(f);

{SI+}

OpenFile := IOResult=0

end; {OpenFile}

Function GetWord: String;

{Получает из файла очередное слово}

var

c: Char;

w: String;

```

Function Letter(var c: Char): Boolean;
{Возвращает TRUE, если символ - буква}
begin
  c := UpCase(c);
  {проверяем на строчную русскую букву:}
  if c in ['a'..'п'] then {a - русская буква}
    c := chr(ord(c)-ord('a')+ord('A')) {A - русская буква}
  else if c in ['р'..'я'] then {р - русская буква}
    c := chr(ord(c)-ord('р')+ord('P')); {P - русская буква}
  {Проверяем на заглавную букву:}
  Letter := c in ['A'..'Z', 'А'..'Я']
end; {Letter}
begin {GetWord}
  w := '';
  c := #0;
  while not EOF(f) and not Letter(c) do
    Read(f,c);
  if not EOF(f) then while not EOF(f) and Letter(c) do
    begin
      w := w+c;
      Read(f,c)
    end;
  GetWord := w
end; {GetWord}
Procedure PrintList(List: PStringCollection);
{Выводит на экран список слов}
Procedure PrintWord(p: PString); far;
begin
  Write(p^, ' ':20-Length(p^))
end; {PrintWord}
begin {PrintList}
  WriteLn;
  WriteLn;
  List^.ForEach(@PrintWord);
  WriteLn
end; {PrintList}
var
  WordList: PStringCollection;
  w: String;
begin {Основная программа}
  if not OpenFile(w) then
    WriteLn('Нельзя открыть файл '+w)
  else
    begin
      WordList := New(PStringCollection, Init(200,10));
      repeat
        w := GetWord;

```

```

    if (w <> '') and (MaxAvail > 255) then
      WordList^.Insert(NewStr(w))
    until w='';
    PrintList(WordList)
  end
end.

```

Отметим, что в операторе

```

    if (w <> '') and (MaxAvail > 255) then

```

осуществляется контроль за доступной динамической памятью. В Turbo Vision есть и встроенные способы контроля кучи — см. п.19.6.

Как и в любой другой отсортированной коллекции, в коллекции строк по умолчанию хранятся элементы с уникальными ключевыми полями. Чтобы подавить контроль за уникальностью строк, добавьте оператор

```

WordList^.Duplicates := True;

```

сразу за оператором создания коллекции

```

WordList := New(PStringCollection, Init(200,10));

```

и сделайте еще один прогон программы, — Вы увидите, как много раз встречается в файле одно и то же слово.

Метод *TStringCollection.Compare* следует перекрыть, если Вы хотите осуществить свой способ сортировки строк. Например, используя объект

type

```

PStrSor = ^TStrSor;

```

```

TStrSor = object (TStringCollection)

```

```

  Function Compare(k1, k2: Pointer): Integer; Virtual;

```

```

end;

```

```

Function TStrSor.Compare(k1, k2: Pointer): Integer;

```

```

var

```

```

  s1: PString absolute k1;

```

```

  s2: PString absolute k2;

```

```

begin

```

```

  if s1^ < s2^ then

```

```

    Compare := 1

```

```

  else if s1^ = s2^ then

```

```

    Compare := 0

```

```

  else

```

```

    Compare := -1

```

```

end;

```

вместо *PStringCollection*, Вы сможете вывести на экран список слов, отсортированных в обратном порядке.

19.6. ПОЛИМОРФНЫЕ КОЛЛЕКЦИИ

Как уже говорилось, коллекции Turbo Vision обладают свойством полиморфизма — они позволяют хранить различные объекты. Поскольку каждый объект имеет все необходимые для него поля и методы, работа с полиморфными коллекциями не создает дополнительных проблем. Действительно, в полиморфной коллекции Вам обычно нет нужды следить за тем, какого типа объект хранится в том или ином элементе — достаточно вызвать нужный виртуальный метод, чтобы осуществить над элементом требуемые действия.

Рассмотрим следующий пример. Пусть необходимо создать и использовать библиотеку графических примитивов (точки, окружности, прямоугольники и т.п.). Каждый из этих элементов может отличаться своим набором полей и методов. Однако некоторые методы выполняют над объектами однотипные действия, такие, например, как создание нового объекта или его вычерчивание на экране. Если эти методы сделать виртуальными и инкапсулировать в объект-предок, каждый из его потомков сможет осуществить нужные действия одинаковым способом.

Для нашего примера можно создать следующий объект-родитель:

type

```
PGraphObject = ^TGraphObject;
TGraphObject = object (TObject)
  X, Y: Integer;           {Координаты характерной точки}
  Constructor Init;        {Создание объекта}
  Procedure Draw; Virtual; {Вычерчивание}
end;
```

Объект *TGraphObject* содержит общие для всех потомков поля и методы. Заметим, что методы *Init* и *Draw* должны перекрываться в объектах-потомках, поэтому их содержимое не имеет значения. Однако полезно вынести в них некоторые общие для всех потомков части программы. Например, конструктор *Init* может помещать в поля *X* и *Y* заданные начальные значения; если этот метод наполнить конкретным содержанием, он может использоваться во всех объектах иерархии:

Constructor TGraphObject.Init;

{Присваивает случайные значения координатам X и Y}

begin

X := Random(GetMaxX);

Y := Random(GetMaxY)

end;

Здесь *GetMaxX*, *GetMaxY* — максимальные координаты графического экрана. Виртуальный метод *Draw* весьма специфичен: его конкретная программная реализация будет существенно зависеть от типа объекта. Поэтому объявим этот метод абстрактным:

Procedure TGraphObject.Draw;

{Абстрактный метод для вычерчивания графического примитива}

begin

```

Abstract
end;

```

Как видим, тело этого метода содержит обращение к глобальной процедуре *Abstract*, которая аварийно завершает выполнение программы и выдает соответствующую диагностику, если в программе используется вызов метода *TGraphObject.Draw*. Таким стандартным способом Turbo Vision сообщает пользователю о некорректности программы. Вы можете сделать тело этого метода другим, если Вас не устраивают стандартные действия, однако во всех случаях имеет смысл предусмотреть возможность некорректного вызова абстрактного метода, даже если вновь создаваемая библиотека будет использоваться только Вами — это значительно облегчит отладку программы.

Создадим три потомка от *TGraphObject*:

```

type
  PPoint = ^TPoint;
  TPoint = object (TGraphObject)    {Точка}
    Procedure Draw; Virtual;
  end;
  PCircle = ^TCircle;               {Окружность}
  TCircle = object (TGraphObject)
    R: Integer;
    Constructor Init;
    Procedure Draw; Virtual;
  end;
  PRectangle = ^TRectangle;         {Прямоугольник}
  TRectangle = object (TGraphObject)
    W, H: Integer;
    Constructor Init;
    Procedure Draw; Virtual;
  end;

```

Объект *TPoint* (точка) не имеет новых полей и поэтому лишь переопределяет абстрактный метод *TGraphObject.Draw*:

```

Procedure TPoint.Draw;
  'Выводит точку на экран'
begin
  PutPixel(X, Y, White)
end;

```

В объектах *TCircle* (окружность) и *TRectangle* (прямоугольник) инкапсулированы новые поля, поэтому переопределяются также и методы *Init*:

```

Constructor TCircle.Init;
  'Создает окружность случайного радиуса в случайном месте'
begin
  TGraphObject.Init;           {Получаем координаты центра}
  R := Random(GetMaxY div 2)  {Получаем радиус}

```



```

end;
Procedure TCircle.Draw;
  {Вычерчивает окружность}
begin
  Circle(X, Y, R)
end;
Constructor TRectangle.Init;
  {Создает случайный прямоугольник}
begin
  TGraphObject.Init;           {Верхний левый угол}
  W := Random(GetMaxX div 2); {Ширина}
  H := Random(GetMaxY div 2) {Высота}
end;
Procedure TRectangle.Draw;
  {Вычерчивает прямоугольник}
begin
  Rectangle(X, Y, X+W, Y+H)
end;

```

После того как определены нужные объекты, не составляет особого труда поместить эти объекты в коллекцию и вывести их на экран. Например, для вывода всех элементов коллекции можно использовать такую процедуру:

```

Procedure DrawAll(C: PCollection);
  {Выводит все элементы полиморфной коллекции}
Procedure DrawItem(p: PGraphObject); far;
begin
  p^.Draw   {Это и есть полиморфизм в действии!}
end;
begin
  C^.ForEach(@DrawItem)
end;

```

Как видим, в процедуре *DrawItem* полиморфизм используется дважды: во-первых, метод *ForEach* обращается к ней, передавая в качестве параметра обращения нетипизированный указатель на элемент коллекции; это позволяет трактовать параметр как указатель на любой объект, в том числе и на *TGraphObject*. Во-вторых, в процедуре используется обращение к виртуальному методу объекта-родителя *Draw*: поскольку этот метод перекрывается во всех потомках, каждый из них будет использовать свой метод *Draw* для вывода на экран.

Сформируем программу, поместив в нее вместо точек уже рассмотренные фрагменты:

```

Uses Objects, Graph, CRT;
type
  .....
Constructor TGraphObject.Init;

```

```
.....
Procedure TGraphObject.Draw;
.....
Constructor TPoint.Init;
.....
Procedure TPoint.Draw;
.....
Constructor TCircle.Init;
.....
Procedure TCircle.Draw;
.....
Constructor TRectangle.Init;
.....
Procedure TRectangle.Draw;
.....
Procedure DrawAll(C: PCollection);
.....
var
  a, r, k: Integer;
  List: PCollection;
  p: Pointer;
begin
  a := 0;
  {Иницилируем графический режим работы экрана:}
  InitGraph(a, r, '\TP\BGI');
  r := GraphResult;
  if r <> 0 then
    WriteLn(GraphErrorMsg(r)) {Ошибка инициации}
  else
    begin
      {Создаем коллекцию:}
      List := New(PCollection, Init(20,5));
      {Наполняем ее 20 элементами:}
      for k := 1 to 20 do
        begin
          case k mod 3 of
            0: p := New(PPoint, Init);
            1: p := New(PCircle, Init);
            2: p := New(PRectangle, Init)
          end;
          if p <> NIL then
            List^.Insert(p)
          end;
        end;
      end;
    end;
  end;

```

```
DrawAll(List);           {Выводим на экран все элементы}
While not KeyPressed do; {Ждем нажатия на любую клавишу}
CloseGraph               {Возвращаемся в текстовый режим}
end
end.
```

В этой программе предполагается, что драйвер графического экрана расположен в каталоге `\TP\BGI` на текущем диске. Если это не так, следует указать маршрут поиска этого драйвера в качестве параметра обращения к процедуре *InitGraph*. Кроме того, каталог, содержащий стандартную графическую библиотеку *Graph*, должен быть указан опцией *Options/Directories/Unit directories*, если, разумеется, библиотека не содержится в текущем каталоге.

19.7. КОЛЛЕКЦИИ И УПРАВЛЕНИЕ ПАМЯТЬЮ

Поскольку элементы коллекций располагаются в динамической памяти, при их использовании особенно важными становятся вопросы контроля за состоянием кучи.

Любая коллекция не может расти до бесконечности: с одной стороны, ее размеры определяются доступной памятью и размером элементов, с другой стороны — общее количество элементов коллекции не может превышать

$$65520 \text{ div } \text{SizeOf(Pointer)} = 16380$$

Иными словами, все указатели на элементы коллекции должны располагаться в пределах одного сегмента. Величина 16380 задается значением глобальной константы *MaxCollectionSize*, объявленной в интерфейсной части модуля *Objects*.

Таким образом, при наполнении коллекции необходимо следить за общим количеством элементов (переменная *TCollection.Count*), которое не может превысить значение, задаваемое константой *MaxCollectionSize*. Кстати, если при обращении к методу *TCollection.Init* начальное значение *N0* коллекции указано слишком большим, оно заменяется на *MaxCollectionSize*.

Элементы коллекции обычно размещаются в куче, поэтому перед размещением очередного элемента следует проверить доступную память (возвращается стандартной функцией *MaxAvail*).

В ходе расширения коллекции может оказаться, что динамической памяти не хватает для размещения нового массива указателей (напомним, что расширение коллекции заключается в создании нового массива из $N0 + k \cdot ND$ указателей, где $N0$ — начальная длина, ND — шаг наращивания коллекции, $k = 1, 2, \dots$; после этого в новый массив копируется старый массив указателей, а место, выделенное под размещение старого массива, возвращается в кучу). Если обнаружена нехватка памяти для расширения коллекции или если при обращении к методу *TCollection.At* указан индекс, превышающий размер коллекции, вызывается метод *TCollection.Error*. По умолчанию этот метод завершает выполнение программы с кодом 212 ошибки периода исполнения. Вы можете перекрыть *TCollection.Error*, чтобы

нужным образом отреагировать на возникшую ситуацию. В этом случае следует учесть, что заголовок *TCollection.Error* в Turbo Vision имеет следующий вид:

Procedure TCollection.Error(Code, Info: Integer);

Параметр *Code* в стандартных ситуациях соответствует следующим константам:

- *coIndexError* = -1 — индекс метода *At* превысил значение *Count* - 1; параметр *Info* содержит неверный индекс;
- *coOverflow* = -2 — нет памяти для расширения коллекции; параметр *Info* содержит требуемый размер коллекции.

Константы *coXXXX* определены в интерфейсной части модуля *Objects*.

Глава 20

ПОТОКИ

Основным способом хранения данных в Turbo Vision является их размещение в полях объектов. Это, разумеется, не означает, что Ваша программа должна использовать исключительно поля объектов и/или методы доступа к ним — программа в среде Turbo Vision — это прежде всего программа на Турбо Паскале, а следовательно, Вы можете использовать в ней и все другие средства этой системы программирования. Однако в ходе изучения Turbo Vision Вы уже, очевидно, смогли по достоинству оценить мощные возможности, предоставляемые техникой объектно-ориентированного программирования и средой Turbo Vision, и, надеюсь, вряд ли захотите широко использовать в своей программе неинкапсулированные данные.

Для сохранения данных на диске с целью последующего их считывания в программу поля можно тем или иным способом выделить из объектов и затем использовать обычные средства доступа к файлам. Согласитесь, что такой прием нельзя считать естественным для объектно-ориентированной программы. Во всяком случае следует учесть, что потоки — а их изучению посвящена эта глава — разработаны прежде всего для сохранения в файлах инкапсулированных данных, т.е. полей объектов.

20.1. СУЩНОСТЬ ПОТОКОВ

Поток Turbo Vision — это коллекция объектов со специфичным способом хранения элементов: для этих целей коллекция-поток использует дисковый файл, отображаемую память (EMS-память) или файловое устройство (устройство ввода-вывода данных).

Весьма удобной моделью потока может служить обычный дисковый файл с прямым или последовательным доступом. В дальнейшем, говоря о потоках, я буду в основном иметь в виду эту модель, а там, где это необходимо, уточнять детали доступа к EMS-памяти и к файловым устройствам.

Как известно, в Турбо Паскале могут использоваться типизированные или нетипизированные файлы. При работе с типизированными файлами Турбо Паскаль осуществляет необходимый контроль за типом данных, предупреждая Вас о возможных ошибках; типизированные файлы как правило не обеспечивают высокую скорость доступа к данным. При обращении к нетипизированным файлам контроль возлагается целиком на программиста, но в этом случае обычно существенно увеличивается скорость работы с файлами. Потоки Turbo Vision обеспечивают в известном смысле компромисс между обоими способами хранения информации в файле: они позволяют компилятору осуществить проверку типов на уровне проверки объектов, но не контролируют сами данные, помещаемые в файл, т.к. в правильно сконструированной программе объект сам осу-

ществляет необходимые операции с диском и делает это безошибочно и очень быстро.

Основным отличием потоков от файлов является их полиморфизм. Как и коллекции, потоки Turbo Vision могут быть полиморфными, что означает, что в одном потоке могут сохраняться разные объекты (точнее, их поля). Как программист, знакомый с техникой объектно-ориентированного программирования, Вы должны понимать, что записать в один файл несколько разных объектов не составляет особой проблемы — для этого нужно лишь вызвать соответствующий виртуальный метод, который знает, какие поля и в какой последовательности помещаются в файл. Но вот каким образом поддерживается полиморфизм при чтении данных? Как из общего потока выделяются данные, относящиеся к конкретному объекту? Эту проблему еще можно было бы решить, если бы потоки обеспечивали только последовательный доступ к данным — тогда мы могли бы просто вызывать виртуальные методы чтения данных в том же порядке, в каком объекты записывались в файл. Но потоки Turbo Vision предоставляют также и произвольный способ доступа к данным!

Решение проблемы разработчиками Turbo Vision найдено в особом механизме, называемом регистрация объектов. Зарегистрировать объект — это означает приписать объекту некоторый уникальный регистрационный номер. Регистрационный номер записывается в файл (а следовательно и считывается из него) первым. Прочитав регистрационный номер, Turbo Vision однозначно определяет объект, которому принадлежат данные, и вызывает соответствующий виртуальный метод доступа к файлу.

Таким образом, для работы с потоками необходимо:

- зарегистрировать объекты, которые будут помещаться в поток или считываться из него; отметим, что все стандартные объекты Turbo Vision уже имеют уникальные регистрационные номера и процедура их регистрации предельно проста (см. ниже);
- инициировать (создать) поток; в ходе инициации указывается имя файла и некоторая другая информация, используемая для обеспечения доступа к файлу;
- поместить данные в поток и/или прочитать их из него;
- удалить поток.

Важной особенностью доступа к потокам является возможность записи в них групп и соответственно чтения групп из потока. При этом группа сама будет автоматически вызывать нужную процедуру доступа для каждого из своих элементов, что значительно упрощает работу с потоком.

20.2. РЕГИСТРАЦИЯ ОБЪЕКТОВ

Для регистрации любого объекта используется обращение к глобальной процедуре *RegisterType*, определенной в интерфейсной части модуля *Objects*. Единственным параметром обращения к этой процедуре является запись типа *TStreamRec*, в которой группируются некоторые важные для Turbo Vision характеристики объекта:

type

```

PStreamRec = ^TStreamRec;
TStreamRec = record
  ObjType: Word;      {Регистрационный номер объекта}
  VMTLink: Word;      {Смещение таблицы виртуальных методов}
  Load   : Pointer;   {Адрес метода чтения}
  Store   : Pointer;   {Адрес метода записи}
  Next    : Word;      {Связь в списке}
end;

```

Для каждого объекта Вашей программы (как стандартного, так и не — стандартного) должна быть создана своя запись типа *TStreamRec*, если только Вы собираетесь помещать соответствующий объект в поток или получать его из потока. Однако для стандартных объектов такие записи уже существуют и Вам нет нужды создавать их заново: по принятому в Turbo Vision соглашению запись *TStreamRec* для стандартного объекта имеет такое же имя, как имя объекта, с заменой начальной буквы *T* на *R*. Например, для *TWindow* регистрационная запись называется *RWindow*, для *TDialog* — *RDialog* и т.д. Имеет смысл следовать этому соглашению, определяя идентификаторы регистрационных записей для нестандартных объектов.

Итак, для каждого нестандартного объекта Вы должны подготовить запись *TStreamRec*, причем фактически определяются только первые четыре поля этой записи, так как поле *Next* используется для организации связанного списка и заполняется внутри процедуры *RegisterType*.

В поле *ObjType* Вы должны поместить константу — идентификатор объекта. В Turbo Vision константы 0...99 уже используются в стандартных регистрационных записях *RXXXX*, таким образом, если Вы собираетесь использовать стандартные регистрационные записи, в Вашем распоряжении остаются константы в диапазоне от 100 до 65535. Выбор констант из этого диапазона может быть произвольным, однако Turbo Vision требует, чтобы каждый объект характеризовался уникальной константой. Заметим, что Turbo Vision следит за уникальностью регистрационных констант и аварийно завершает исполнение программы (код ошибки 212), если в программе делается попытка зарегистрировать два разных объекта с одинаковыми регистрационными номерами.

Поле *VMTLink* должно содержать смещение адреса таблицы виртуальных методов (ТВМ) объекта. Турбо Паскаль имеет стандартную функцию *TypeOf* (*Type: object*), которая возвращает адрес ТВМ для указанного типа *Type* объекта, поэтому поле *VMTLink* обычно заполняется значением *Ofs (TypeOf (TName)^)*, где *TName* — имя нестандартного объекта.

Поля *Load* и *Store* должны содержать адреса соответствующих виртуальных методов (см. п.20.3), обеспечивающих чтение объектов из потока и их запись в поток.

Например, если в программе определен объект

```

type
  TMyWindow = object (TWindow)
    .....

```

```

Constructor Load(var S: TStream);
Procedure Store(var S: TStream);
end;

```

то для его регистрации создается запись

```

const
  RMyWindow: TStreamRec = (
    ObjType: 100;
    VMTLink: Ofs (TypeOf (TMyWindow) ^);
    Load   : @TMyWindow.Load;
    Store   : @TMyWindow.Store);

```

Обычно регистрация осуществляется в конструкторе *Init* вновь созданного объекта, например:

```

Constructor TMyWindow.Init;
begin
  RegisterType (RMyWindow);
  .....
end;

```

Однако, если в программе предполагается помещать в поток (или считывать из потока) объекты разного типа, регистрацию объектов как правило реализуют отдельной процедурой, вызываемой в конструкторе *Init* программы или в любом другом удобном месте, но перед фактическим обращением к потоку. Например:

```

Procedure RegisterMyProgram;
begin
  RegisterType (RWindow); {Регистрация стандартного объекта}
  RegisterType (RMyWindow); {Регистрация нового объекта}
  .....
end;

```

Для упрощения регистрации стандартных объектов в модулях Turbo Vision предусмотрены процедуры *RegisterXXXX*, где XXXX — имя соответствующего модуля. Например, процедура *RegisterDialogs* осуществляет регистрацию всех неабстрактных объектов модуля *Dialogs*, т.е. *TButton*, *TCluster*, *TInputLine* и т.д.; процедура *RegisterViews* регистрирует объекты модуля *Views* и т.д.

20.3. СОЗДАНИЕ И УДАЛЕНИЕ ПОТОКА

Для работы с потоками в Turbo Vision предусмотрен абстрактный тип *TStream* и три его потомка — *TDOStream*, *TBufStream* и *TEMSSStream*.

Объект *TDOStream* реализует небуферизованный доступ к файлу или файловому устройству. Буферизация потока означает использование некоторого промежуточного буфера, в который помещаются данные перед физической записью их на диск или сразу после физического чтения с диска. Буферизация позволяет согласовать формат данных с размерами

дискового сектора и обычно значительно ускоряет доступ к потоку, поэтому в большинстве случаев в программах используется буферизованная версия потока, реализуемая объектом *TBufStream*.

Объект *TEMSSStream* обеспечивает доступ к отображаемой памяти компьютера, оборудованного специальной *EMS*-платой (для ПК с процессорами 80386 и более поздними *EMS*-память может эмулироваться). Запись объектов в *EMS*-память и чтение их из нее осуществляется с предельно возможной скоростью, однако содержимое этой памяти разрушается после выключения компьютера. Таким образом, *TEMSSStream* используется для временного хранения данных с целью минимизации времени доступа к ним. Если в программе предполагается частое обращение к потоку, имеет смысл скопировать его в *EMS*-память в начале работы программы и перенести хранящиеся в нем данные на диск перед завершением работы.

В каждом из объектов *TXXXStream* предусмотрен свой конструктор *Init*, с помощью которого создается экземпляр нужного потока. Ниже описывается формат вызова каждого конструктора.

Constructor *TDOSStream.Init(FileName: FNameStr; Mode: Word);*

Здесь *FileName* — имя дискового файла или файлового устройства, *Mode* — способ доступа к данным.

Параметр *FileName* может содержать полное имя файла с указанием диска и маршрута поиска. Параметр *Mode* определяет способ доступа к данным. Для задания этого параметра можно использовать следующие константы, определенные в модуле *Objects*:

const

<i>stCreat</i>	=	<i>\$3C00</i> ;	{Создать файл}
<i>stOpenRead</i>	=	<i>\$3D00</i> ;	{Открыть файл только для чтения}
<i>stOpenWrite</i>	=	<i>\$3D01</i> ;	{Открыть файл только для записи}
<i>stOpen</i>	=	<i>\$3D02</i> ;	{Открыть файл для чтения и записи}

Constructor *TBufStream(FileName: FNameStr;
Mode: Word; Size: Word);*

Здесь *FileName*, *Mode* — см. выше; *Size* — размер буфера в байтах.

Размер буфера выбирается равным размеру дискового сектора (512 байт) или размеру кластера ($n \cdot 512$, $n = 1, 2, 4, 8, \dots$). Минимальные потери времени обеспечивает размер буфера, равный размеру кластера диска (кластер — минимальная порция дискового пространства, выделяемая каждому файлу). При работе с гибкими дисками размер кластера обычно равен одному или двум секторам, для жесткого диска этот размер зависит от общей емкости диска и чаще всего равен 4 или 8 секторам. Если Вы не знаете размеры кластера диска, с которым будет связан поток, установите *Size = 512*.

Constructor *TEMSSStream.Init(MinSize, MaxSize: LongInt);*

Здесь *MinSize*, *MaxSize* определяют соответственно минимальный и максимальный размеры блока, который будет передаваться в *EMS*-память.

Параметр *MaxSize* имеет смысл только при использовании драйвера *EMS* — памяти, версии меньше 4.0: в этом случае попытка разместить в расширенной памяти блок, больше *MaxSize*, вызовет ошибку; при использовании версии драйвера 4.0 и выше в памяти можно разместить блок любого размера, в этом случае параметр *MaxSize* можно опускать.

После завершения работы с потоком следует удалить его экземпляр — это аналогично тому, как Вы закрываете дисковый файл после его использования. Для удаления потока нужно обратиться к его методу *Done*, например:

```
Dispose(PMyStream, Done);
```

Здесь *PMyStream* — указатель на экземпляр потока, размещенный в куче. Если Вы не использовали указатель на поток (т.е. если экземпляр потока размещен в обычной переменной), для удаления потока используется вызов

```
MyStream.Done;
```

(*MyStream* — экземпляр потока).

В ходе реализации процедуры *TStream.Done* очищается внутренний буфер (если использовался буферизованный поток), закрывается файл и уничтожается экземпляр потока.

20.4. РАБОТА С ПОТОКОМ

Базовый объект *TStream* реализует три метода, используемых для непосредственной работы с потоком. Метод *TStream.Put* предназначен для передачи объектов в поток и выполняет приблизительно такие же функции, как стандартная файловая процедура *Write*. Метод *TStream.Get* используется для чтения объектов из потока, его аналогом является процедура *Read*. Наконец, с помощью метода *TStream.Error* анализируется состояние потока после завершения некоторой операции: если обнаружена ошибка при обмене данными с потоком, вызывается этот метод, который по умолчанию просто устанавливает признаки ошибки в информационных полях *TStream.Status* и *TStream.ErrorInfo*. Приблизительным аналогом метода *TStream.Error* служит стандартная файловая функция *IOResult*.

Сразу же замечу, что в случае возникновения ошибки все последующие операции с потоком блокируются до тех пор, пока не будет вызван метод *TStream.Reset*.

Методы *Put* и *Get* практически никогда не перекрываются: для реализации операций с потоком они обращаются к виртуальным методам *Store* и *Load*, которые должны быть определены в каждом объекте, если только этот объект помещается в поток или считывается из него. Главное назначение методов *Put* и *Get* состоит в обеспечении полиморфизма потока за счет контроля регистрационных номеров объектов. Методы *Load* и *Store* никогда не вызываются прямо, но только из методов *Put* и *Get*, т.к. они ничего не знают о регистрационных номерах и не могут работать в полиморфных потоках.

20.4.1. Методы Put и Get

Чтобы поместить объект в поток, нужно обратиться к методу *Put*, передав ему в качестве параметра инициированный экземпляр объекта. Например:

```
var
  MyStream: TBufStream;      {Экземпляр потока}
  MyWindow: TMyWindow;      {Экземпляр объекта}
.....
  MyStream.Put(MyWindow);    {Помещаем объект в поток}
```

Предварительно объект должен быть зарегистрирован обращением к *RegisterType*, а поток — иницирован с помощью *TXXXStream.Init*.

Метод *Put* вначале отыскивает объект в регистрационном списке, создаваемом процедурой *RegisterType*, и получает из этого списка регистрационный номер объекта и адрес его метода *Store*. Затем в поток записывается регистрационный номер и вызывается метод *Store*, который делает остальное, т.е. копирует в поток все поля объекта.

По такой же схеме работает и метод *Get*: вначале он считывает из потока регистрационный номер объекта, затем отыскивает его в регистрационном списке и вызывает соответствующий конструктор *Load*. Конструктор размещает в динамической памяти экземпляр считываемого объекта, а затем считывает из потока все его поля. Результатом работы *Get* является нетипизированный указатель на вновь созданный и инициированный объект. Например:

```
type
  MyStream: TBufStream;      {Экземпляр потока}
  PWindow: PMyWindow;       {Указатель на экземпляр объекта}
.....
  PWindow := MyStream.Get;   {Получаем объект из потока}
```

Заметим, что количество считываемых из потока данных и тип *TBM*, который назначен вновь созданному объекту, определяется не типом *PWindow* (см. выше), а регистрационным номером, полученным из потока. Вы можете ошибочно поместить в левой части оператора присваивания указатель на объект другого типа и Turbo Vision не сможет предупредить Вас об этом!

Методы *Put* и *Get* позволяют автоматически сохранять в потоке и получать из него сложные объекты (группы). Эта возможность реализуется внутри методов *Store* и *Load*.

20.4.2. Методы Store и Load

Метод *Store* осуществляет запись данных в поток. Для этого он использует метод низкого уровня *Write*, передавая ему в качестве параметров имя записываемого поля и длину поля в байтах. Заметим, что Вам нет нужды записывать все поля объекта: для записи наследуемых пслей достаточно просто обратиться к методу *Store* объекта-родителя. Ваш

метод *Store* должен записывать только те поля, которые добавляются к полям родителя. Если, например, создан объект

```
type
  TMyDialog = object (TDialog)
    St: String;                                {Новое поле}
    Procedure Store(var S: TStream); Virtual;
    .....
  end;
```

то метод *TMyDialog.Store* может иметь такую реализацию:

```
Procedure TMyDialog.Store(var S: TStream);
begin
  TDialog.Store(S);                            {Сохраняем наследуемые поля}
  S^.Write(St, SizeOf(St));                    {Сохраняем новое поле}
end;
```

Аналогичным образом реализуется и конструктор *Load*: с помощью обращения к низкоуровневому методу *TStream.Read* он получает из потока только дополнительные поля и только в том порядке, как они были записаны в поток методом *Store*:

```
Constructor TMyDialog.Load(var S: TStream);
begin
  TDialog.Load(S);                            {Получаем наследуемые поля}
  S^.Read(St, SizeOf(St));                    {Получаем новое поле}
end;
```

Вы должны тщательно следить за соответствием методов *Store* и *Load*: метод *Load* должен прочитать ровно столько байт и строго в той последовательности, сколько байт и в какой последовательности поместил в поток метод *Store*. В Turbo Vision нет средств контроля за правильностью считываемых данных!

Если Ваш объект — группа, следует включить в него поля-указатели на каждый из элементов и использовать методы *PutSubviewPtr* и *GetSubviewPtr* соответственно для записи в поток и чтения из него. Например:

```
type
  TMyDialog = object (TDialog)
    St: String;                                {Текстовое поле}
    PB: PButton;                              {Указатель на кнопку}
    Procedure Store(var S: TStream); Virtual;
    Constructor Load(var S: TStream);
    .....
  end;
Procedure TMyDialog.Store(var S: TStream);
begin
  TDialog.Store(S);                            {Сохраняем наследуемые поля}
  S^.Write(St, SizeOf(St));                    {Сохраняем текстовое поле}
```

```

PutSubViewPtr(S, PB);      {Сохраняем кнопку}
end;
Constructor TMyDialog.Load(var S: TStream);
begin
  TDialog.Load(S);          {Получаем наследуемые поля}
  S^.Read(St, SizeOf(St));  {Получаем тестовое поле}
  GetSubViewPtr(S, PB);     {Получаем кнопку}
end;

```

20.4.3. Обработка ошибок

При обнаружении ошибки поток вызывает свой метод *TStream.Error*, который определяет необходимую реакцию программы. По умолчанию этот метод просто записывает информацию об ошибке в поля *TStream.Status* и *TStream.ErrorInfo*.

Поле *Status* определяет тип ошибки, в соответствии со следующими константами модуля *Objects*:

```

const
  stOk          = 0; {Нет ошибки}
  stError       = -1; {Ошибка доступа}
  stInitError   = -2; {Ошибка инициации потока}
  stReadError   = -3; {Чтение за концом потока}
  stWriteError  = -4; {Нельзя расширить поток}
  stGetError    = -5; {Get для незарегистрированного объекта}
  stPutError    = -6; {Put для незарегистрированного объекта}

```

Поле *ErrorInfo* определено только для *Status* = -5 или *Status* = -6: в первом случае оно содержит регистрационный номер, полученный из потока и не обнаруженный в регистрационном списке; во втором — смещение ТВМ незарегистрированного объекта, который программа пытается поместить в поток.

Сразу после обнаружения ошибки Turbo Vision блокирует все операции с потоком до тех пор, пока аварийная ситуация не будет сброшена с обращением к методу *TStream.Reset*.

20.4.4. Прямой доступ к потокам

Поток имеет методы, имитирующие файловые процедуры прямого доступа к дисковому файлу.

С помощью функции *GetPos* программа может получить текущую позицию в потоке, т.е. номер байта, начиная с которого будет осуществляться очередная операция с потоком (первый байт потока имеет номер 0).

Метод *Seek (Pos: LongInt)* перемещает текущую позицию в потоке в байт *Pos* от начала потока.

Метод *GetSize* возвращает общий размер потока в байтах.

С помощью метода *Truncate* можно удалить из потока все данные, начиная с текущей позиции до конца потока.

Как видим, эти процедуры можно использовать только в том случае, если создать вне потока индексную коллекцию, содержащую начальные

позиции в потоке для каждого из сохраняемых в нем объектов. Такая коллекция используется в ресурсах, поэтому для обеспечения прямого доступа к потоку лучше использовать файл ресурсов (см. гл.21).

20.4.5. Использование потоков с произвольными данными

Хотя потоки спроектированы в основном для работы с объектами, Вы можете использовать их для хранения не только полей объектов, но и любых других данных. При этом не следует обращаться к методам *Put* и *Get*, так как они предполагают доступ к объектам. Вместо этого Вы должны обратиться к низкоуровневым процедурам *Write* и *Read*.

Следующая простая программа использует поток для сохранения десяти случайных целых чисел:

```
Uses Objects;
var
  S: TBufStream; {Экземпляр потока}
  k,j: Integer;
begin
  WriteLn('Запись в поток:');
  S.Init('Test.dat', stCreate, 512); {Создаем поток}
  for k := 1 to 10 do
    begin
      j := Random(100); {Получаем случайное целое}
      Write(j:8);        {Выводим на экран}
      S.Write(j,2)        {Помещаем в поток}
    end;
  S.Done;                {Удаляем поток}
  S.Init('Test.dat', stOpenRead, 512);
  WriteLn;
  WriteLn('Чтение из потока:');
  for k := 1 to 10 do
    begin
      S.Read(j,2);        {Получаем целое из потока}
      Write(j:8)          {Выводим на экран}
    end;
  S.Done;
  WriteLn
end.
```

Для простоты в программу не включены средства контроля ошибок. В ходе прогона программы в текущем каталоге диска будет создан файл *TEST.DAT* размером в 20 байт, а на экран будут выведены две строки с одинаковыми числами.

РЕСУРСЫ TURBO VISION

Ресурсы — это индексированные потоки. Главное отличие ресурсов от потоков заключается в том, что к объектам, хранящимся в ресурсе, можно обращаться по ключам — уникальным строкам, идентифицирующим объекты. Таким образом, ресурсы спроектированы специально для облегчения произвольного доступа к потокам.

Использование ресурсов открывает перед Вами ряд интересных возможностей.

Вы можете загружать любые видимые элементы из предварительно созданного файла ресурсов вместо того, чтобы создавать их в теле основной программы. Так как реализация методов *Load* обычно значительно проще, чем реализация конструкторов *Init*, это дает возможность не только уменьшить размер Вашей программы, поскольку в основной программе Вы уже не описываете методы *Init*, а просто загружаете нужные объекты из ресурса. Разумеется, Вы должны предварительно разработать вспомогательную программу, создающую нужный файл ресурсов.

Разные ресурсы могут хранить одинаковое количество текстовых строк. Поскольку к каждой строке ресурса можно обращаться по индексу, Вы можете простой заменой файла ресурсов изменить, например, язык, на котором написаны сообщения встроенной справочной службы. Причем такая замена никак не отражается на содержательной стороне справок и не влечет за собой никаких изменений в тексте программы. Все, что Вам необходимо в этом случае, — это подготовить несколько идентичных по смыслу, но написанных на разных языках файлов текстовых ресурсов. Для отечественных программистов, ориентирующихся на западный рынок, такая возможность, согласитесь, может показаться весьма заманчивой.

Наконец, смена файла ресурсов позволит Вам легко создать демонстрационную версию Вашей программы без какой-либо ее перенастройки. Для этого Вы готовите два файла ресурсов: один рассчитан на полные возможности программы, другой позволяет использовать только часть ее возможностей и предоставляет пользователю сокращенные меню и строки статуса. Поставка программы с тем или иным файлом ресурсов определяет ее возможности.

В Turbo Vision ресурс реализуется с помощью объекта *TResourceFile*, который содержит поток и связанную с ним отсортированную коллекцию строк. С помощью метода *Init* Вы создаете ресурс, методом *Put* помещаете в поток нужные Вам объекты, а с помощью метода *Get* получаете из него объект по его имени или индексу.

21.1. СОЗДАНИЕ РЕСУРСА

Поскольку ресурс — это поток, для создания ресурса необходимо открыть (создать) поток. Для этого следует использовать метод *Init* для потока нужного типа (обычно используется буферизованный поток *TBufStream* или

его потомок). После того как поток успешно открыт, в него помещают нужные объекты с их уникальными ключами и затем поток закрывается — файл ресурсов готов.

Вот как, например, можно создать ресурс, содержащий строку статуса с именем (ключом) «Статус»:

```

Uses Drivers, Objects, Views, App, Menu;
var
  StatusRes: TResourceFile;    {Файл ресурсов}
  PStatusStrm: PBufStream;    {Буферизованный поток}
Procedure CreateStatusLine;
  {Создает строку статуса и помещает ее в файл ресурсов}
var
  R: TRect;
  PStatus: PStatusLine;
begin
  R.Assign(0,24,80,25);
  PStatus := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Выход', kbAltX, cmQuit,
        NewStatusKey('Демонстрация строки статуса',0,0, NIL)),
      NIL)));
  StatusRes.Put(PStatus, 'Статус');
  Dispose(PStatus, Done)
end;
begin
  PStatusStrm := New(PBufStream, Init(
    'status.res', stCreate, 1024));
  StatusRes.Init(PStatusStrm); {Создаем поток}
  RegisterType(RStatusLine);   {Регистрируем строку статуса}
  CreateStatusLine;            {Помещаем строку статуса в поток}
  StatusRes.Done
end.

```

В программе создается поток, связанный с указателем *PStatusStrm*, и в него с помощью процедуры *CreateStatusLine* помещается строка статуса. Этой строке присваивается ключ *Статус*.

21.2. ИСПОЛЬЗОВАНИЕ РЕСУРСА

Получить объект из ресурса не представляет проблемы: после инициализации файла ресурсов Вы просто используете его метод *Get*, которому передаете ключ (имя) нужного ресурса. Например, в следующей программе используется файл *Status.res*, созданный в предыдущем примере:

```

Uses Objects, Drivers, Views, Menu, Dialogs, App;
var
  StatusRes: TResourceFile;    {Файл ресурсов}
type
  PMyApp = ^TMyApp;           {Программа в Turbo Vision}

```



```
TMyApp = object (TApplication)
  Constructor Init;
  Procedure InitStatusLine; Virtual;
end;
Constructor TMyApp.Init;
{Открывает поток, связанный с файлом ресурсов}
begin
  StatusRes.Init(New(PBufStream,
    Init('status.res',stOpen,1024)));
  if StatusRes.Stream^.Status <> 0 then
    Halt;
  RegisterType(RStatusLine);
  Inherited Init;
end;
Procedure TMyApp.InitStatusLine;
{Получает строку статуса из файла ресурсов}
begin
  StatusLine := PStatusLine(StatusRes.Get('Статус'))
end;
var
  Prog: TMyApp;
begin
  Prog.Init;
  Prog.Run;
  Prog.Done
end.
```

Заметим, что если в файле ресурсов нет ресурса с указанным именем, метод *Get* возвращает значение *NIL*.

Любой ресурс можно получить из файла ресурсов сколько угодно раз и в любом порядке, т.е. файл ресурсов — это поток с произвольным доступом. При необходимости можно дополнять ранее созданный файл ресурсов новыми объектами или заменять существующие в нем объекты другими.

Для ускорения доступа к файлу ресурсов его можно связать с EMS-памятью или использовать для его размещения виртуальный диск.

21.3. СТРОКОВЫЕ РЕСУРСЫ

Списки строк используются в программах Turbo Vision очень часто, поэтому специально для этого вида ресурсов разработаны два особых объекта — *TStrListMaker* и *TStringList*. С помощью *TStrListMaker* создается строковый ресурс, а с помощью *TStringList* ранее созданный строковый ресурс становится доступен программе. Предполагается, что создание строкового ресурса и его использование осуществляется в разных программах, поэтому обоим объектам в Turbo Vision присвоен один и тот же регистрационный номер. По той же причине *TStrListMaker* имеет метод *Put*, но не имеет метода *Get*, а *TStringList*, наоборот, имеет *Get* и не имеет *Put*.

В отличие от остальных ресурсов доступ к строковым ресурсам осуществляется по индексам. Таким образом, Ваша программа становится независимой от языка, используемого для вывода сообщений. Чтобы поместить на экран ту или иную информацию, достаточно указать индекс нужного сообщения и в зависимости от используемого файла строкового ресурса сообщение будет выведено на нужном языке.

В двух следующих программах иллюстрируется создание и использование строкового ресурса. В программе *CreateStringRes* создается строковый ресурс, содержащий все строки текста программы. Константы *NStr* и *SizeStrRes* определяют количество строк и общую длину строкового ресурса в байтах. В нашем примере эти величины заранее не известны и выбираются приблизительно, но так, чтобы в ресурсе гарантированно разместился бы весь текст программы. В реальной программе они должны выбираться, исходя из конкретной задачи.

Program CreateStringRes;

{Эта программа создает строковый ресурс}

Uses Objects;

const

SizeStrRes = 2000; {Общая длина всех строк в ресурсе}
NStr = 40; {Общее количество строк}

var

f: text; {Файл с текстом программы}
ProgText: TResourceFile;
Strings : TStrListMaker; {Строковый ресурс}
k: Integer;
s: **String**;

begin

s := copy(ParamStr(0),1,pos('.',ParamStr(0)));
Assign(f,s+'PAS');
Reset(f); {Открываем файл с текстом программы}
RegisterType(RStrListMaker); {Регистрируем объект}
ProgText.Init(New(PBufStream,
Init('prog.res',stCreate,1024)));
Strings.Init(SizeStrRes,Nstr);
k := 0;
while not EOF(f) **do**
 begin
 ReadLn(f,s); {Читаем строку программы}
 inc(k); {k - порядковый номер строки}
 Strings.Put(k,s) {Помещаем строку в ресурс}
 end;
Close(f); {Закрываем файл с текстом программы}
{Помещаем ресурс в поток}
ProgText.Put(@Strings,'Программа');
Strings.Done; {Закрываем ресурс}
ProgText.Done {Закрываем поток}

end.

В программе *UseStringRes* текст, полученный из созданного строкового ресурса, читается и выводится на экран «задом наперед», начиная с последней строки текста.

```
Program UseStringRes;  
{Эта программа использует ранее созданный строковый ресурс}  
Uses Objects;  
var  
  ProgText: TResourceFile; {Файл ресурсов}  
  PStrings : PStringList; {Строковый ресурс}  
  k, N: Integer;  
begin  
  RegisterType(RStringList);           {Регистрируем объект}  
  ProgText.Init(New(PBufStream,         {Создаем поток}  
    Init('prog.res', stOpenRead, 1024)));  
  PStrings :=                          {Получаем из потока ресурс}  
    PStringList(ProgText.Get('Программа'));  
  N := 1;  
  while (PStrings^.Get(N) <> '') do  
    inc(N);                             {N-1 = общее количество строк}  
  for k := N-1 downto 1 do  
    WriteLn(PStrings^.Get(k));          {Получаем и выводим строки}  
  PStrings^.Done;                       {Закрываем ресурс}  
  ProgText.Done                         {Закрываем поток}  
end.
```

Заметим, что индексы, по которым осуществляется доступ к строковому ресурсу, могут быть произвольными целыми числами в диапазоне от 0 до 65535. Чтобы определить общее количество строк в строковом ресурсе, используется то обстоятельство, что при обращении к *TStringList.Get* с недействительным индексом метод выдает пустую строку. На практике этот способ нельзя считать надежным (в строковом ресурсе могут храниться и пустые строки), однако других способов определения этого параметра в *TStringList* не существует.

Перед обращением к *TSrtListMaker.Put* и *TSrtngList.Get* соответствующие объекты должны быть зарегистрированы с помощью *RegisterType*.

ОБЪЕКТЫ–КОНТРОЛЕРЫ

При разработке диалоговых программ часто бывает необходимо не только предоставить пользователю возможность ввода данных, но и проверить их допустимость в данном контексте программы. Если, например, пользователь должен вводить целочисленные величины, то в набранной им строке могут содержаться только цифры, а если от него ожидается ввод имени файла, строка должна представлять собой правильный маршрут поиска дискового файла.

В этой небольшой главе рассматривается имеющаяся в Turbo Vision группа объектов–контролеров, упрощающих решение подобного рода задач.

22.1. ТИПЫ ОБЪЕКТОВ–КОНТРОЛЕРОВ

Объект–контролер обычно связывается со строкой ввода *TInputLine* и активизируется при выполнении метода *TInputLine.Valid*. Получив управление, контролер осуществляет необходимую проверку данных и блокирует завершение ввода, если обнаружена ошибка. В *ObjectWindows* входят пять объектов–контролеров, являющихся наследниками абстрактного объекта *TValidator* и отличающихся алгоритмом проверки данных.

22.1.1. TPXPictureValidator

Объект *TPXPictureValidator* проверяет введенную пользователем строку на соответствие некоторому шаблону ввода. Шаблон ввода определяется с помощью специальных символов, подобно тому, как задается допустимый ввод в базе данных *Paradox* корпорации *Borland*. В шаблоне можно использовать следующие специальные символы:

Символ	Назначение
#	Любая цифра
?	Любая буква
@	Любой символ (буква, цифра или специальный символ)
&	Любая буква с автоматическим преобразованием в верхний регистр
!	Любой символ с автоматическим преобразованием букв в верхний регистр
*	Повторение следующего символа произвольное число раз
;	Следующий символ – не управляющий
[]	Заключенная в скобки последовательность символов может отсутствовать
,	Разделение вариантов
{ }	Создание группы

Любой другой символ шаблона представляет сам себя, т.е. не является управляющим. Символ «;», предшествующий специальному символу, отменяет его специальное назначение. Чтобы в шаблон вставить собственно символ «;», его нужно удвоить.

Перечисленные символы позволяют формировать шаблоны любой сложности. Например:

- ### — трехзначное число; примеры правильного ввода: 123, 566, 080; неправильный ввод: 12 (нет одной цифры), -12 (первый символ — не цифра);
- *# — число с произвольным количеством цифр; допустимый ввод: 1, 12, 12345678987654321; неправильный ввод: -12, 0334A;
- [+,-]*# — произвольное целое число; допустимый ввод: +123, -56, 7890; недопустимый ввод: +-123, 1.23, 12345E+02;
- {#[#]}-{#[#]}-{##[##]} — дата в формате ДД-ММ-ГГ; допустимый ввод: 1-2-33, 01-02-1933; недопустимый ввод: 1/2/33, 1 фев 33;
- #;***# — число в диапазоне 0...9 умножается (символ *) на произвольное положительное целое число; правильный ввод: 2*2, 3*12345; неправильный ввод: 2/2, 123*12345.

22.1.2. TFilterValidator

Этот объект проверяет все введенные пользователем символы на их принадлежность к набору допустимых символов. Если очередной вводимый символ не принадлежит эталонному множеству, он игнорируется и при наборе просто не появляется в строке ввода. Таким образом, объект *TFilterValidator* действительно *фильтрует* вводимую строку, удаляя из нее неправильные символы. Например, экземпляр объекта

```
MyFilter := New(PFilterValidator, Init(['0'..'9']))
```

выбирает из ввода только цифры.

22.1.3. TRangeValidator

Объект *TRangeValidator* порожден от *TFilterValidator*. Он преобразует символьный ввод в целое число и проверяет, находится ли оно в диапазоне указанных значений *Min...Max*. Его метод *IsValid* вернет значение *True* только в том случае, когда, во-первых, пользователь введет правильное число, и, во-вторых, это число будет принадлежать нужному диапазону. Например, объект

```
MyRange := New(PRangeValidator, Init(0, 100))
```

позволяет вводить числа в диапазоне от 0 до 100 включительно.

22.1.4. TLookupValidator

Абстрактный объект *TLookupValidator* предназначен для контроля со-ответствия введенной пользователем строки набору эталонных строк. Фактически он модифицирует поведение базового объекта *TValidator* на случай, когда проверяется произвольная текстовая строка. В его потомках должен перекрываться метод *Lookup*, осуществляющий нужный контроль и возвращающий *True* только в том случае, когда подготовленная пользо-

вателем строка соответствует одному из желаемых значений. Например, с помощью объекта

type

```
  TMyLookupValidator = object(TLookupValidator)
    Function Lookup(const S: String): Boolean; Virtual;
  end;
```

```
Function TMyLookupValidator.Lookup;
const
```

```
  Text1 = 'Turbo';
  Text2 = 'Pascal';
```

begin

```
  Lookup := (S=Text1) or (S=Text2) or (S=Text1+' '+Text2)
```

end;

можно потребовать от пользователя ввода только одной из трех возможных строк: *Turbo*, *Pascal* или *Turbo Pascal*.

22.1.5. TStringLookupValidator

Этот объект порожден от *TLookupValidator* и сравнивает введенную пользователем строку с набором допустимых строк, хранящихся в отсортированной коллекции. Чтобы объект осуществлял сравнение с нужным набором строк, он инициализируется следующим образом:

var

```
  MyCollection: PStringCollection;
  MyStrVal: PStringLookupValidator;
```

begin

```
  {Создаем коллекцию строк}
```

```
  MyCollection := New(PStringCollection, Init(1,1));
```

```
  {Наполняем коллекцию - размещаем строки в куче}
```

```
  MyCollection^.Insert(NewStr('Турбо'));
```

```
  MyCollection^.Insert(NewStr('Паскаль'));
```

```
  {Создаем объект-контролер}
```

```
  MyStrCol := New(PStringLookupValidator, Init(MyCollection))
```

```
  .....
```

end;

22.2. ИСПОЛЬЗОВАНИЕ ОБЪЕКТОВ-КОНТРОЛЕРОВ

22.2.1. Контроль текстового ввода

Как уже говорилось, основное назначение объектов-контролеров состоит в контроле текстовой информации, вводимой пользователем с помощью объектов *TInputLine* или их специализированных потомков. Для упрощения подключения контролера в объекте *TInputLine* предусмотрен специальный метод *SetValidator*, единственным параметром обращения к которому является ссылка на экземпляр объекта-контролера. Таким образом, подключение контролера осуществляется в два этапа: вначале

инициируется экземпляр объекта-контролера, а затем вызывается метод *TInputLine.SetValidator* для объекта, осуществляющего ввод данных. На-пример:

```
var
  InpLine: PInputLine;
  Validator: PRangeValidator;
begin
  {Создаем строку ввода}
  InpLine := New(PInputLine, Init(...));
  {Создаем объект-контролер}
  Validator := New(PRangeValidator, Init(0,10));
  {Связываем контролер с редактором}
  InpLine^.SetValidator(Validator);
  .....
end;
```

Можно объединить оба действия в одном операторе:

```
InpLine^.SetValidator(New(PRangeValidator, Init(0,10)))
```

При работе совместно с объектом *TInputLine* контролер может активно воздействовать на ввод пользователя. Это относится к объектам *TFilterValidator* и *TPXPictureValidator*: они контролируют ввод каждого символа и игнорируют нажатие пользователем клавиши, если очередной символ не соответствует требуемому. Объект *TPXPictureValidator*, кроме того, способен автоматически дополнять ввод пользователя неспециальными символами шаблона. Если, например, шаблон задан следующим образом:

```
InpLine^.SetValidator(New(PPXPictureValidator,
  Init('#/#/#', True)))
```

то нажатие пользователем любых клавиш, кроме цифровых, будет игнорироваться, а сразу после нажатия первой (второй) цифровой клавиши в строке ввода появится символ «/». Вставкой неспециальных символов шаблона управляет второй параметр обращения к методу *TPXPictureValidator.Init* (*True* — разрешить вставку, *False* — запретить).

22.2.2. Проверка других объектов

Хотя чаще всего для реализации текстового ввода Вы будете использовать объект *TInputLine* или его специализированного потомка, не исключена возможность проверки данных, подготовленных другими видими объектами. Для этого в поле *Options* любого видимого элемента предусмотрен бит *ofValidate*, единичное состояние которого приведет к тому, что при каждой потере видимым элементом фокуса ввода будет вызываться его метод *Valid*, который он наследует от *TView*. Перекрыв этот метод, Вы сможете обратиться к любому объекту-контролеру для проверки данных.

22.2.3. Реализация нестандартного контроля

Возможно, Вам понадобится сконструировать собственный объект — контролер. В этом случае Вы должны помнить, что за контроль отвечают четыре метода базового объекта *TValidator*: *Valid*, *IsValidInput*, *IsValid* и *Error*, причем абстрактные методы *IsValid* и *Error* перекрываются в потомках, вызываются другими методами объекта — контролера и не предназначены для вызова извне.

Метод *Valid* получает строку, введенную пользователем, осуществляет ее контроль и возвращает *True* или *False* в зависимости от результатов контроля (*True* означает правильный ввод). Метод *IsValidInput* также получает строку символов ввода, однако, в отличие от *Valid*, он получает ее после каждого изменения пользователем входной строки. Метод *IsValidInput* имеет два параметра обращения:

```
Function TValidator.IsValidInput (var S: String;  
    SuppressFill: Boolean): Boolean
```

Параметр *SuppressFill* указывает, должен ли метод автоматически форматировать строку перед ее проверкой. Если этот параметр имеет значение *True*, метод должен дополнить подготовленный пользователем ввод *S* нужными в этом месте символами и только после этого производить контроль. Поскольку параметр *S* определен как параметр — переменная, метод может вставлять дополнительные символы непосредственно в строку ввода, но ни в коем случае не должен удалять ошибочно введенные символы из нее. Из всех стандартных контролеров только *TPXPictureValidator* использует *SuppressFill = True*.

ПРАКТИКА ИСПОЛЬЗОВАНИЯ

Программирование с использованием библиотеки Turbo Vision весьма специфично по сравнению с обычным программированием в среде Турбо Паскаля. Эта специфика прежде всего связана с широко используемым в Turbo Vision механизмом ООП: как правило, нельзя чисто механически перенести старую программу в новую объектно-ориентированную среду, обычно для этого требуется переосмысление всего проекта в целом. Другая особенность Turbo Vision — интенсивное использование динамической памяти: если Вы не имеете достаточного опыта работы с кучей, у Вас могут быть проблемы, связанные с динамически размещаемыми объектами. И, наконец, программы, управляемые событиями — это еще одна весьма специфичная особенность Turbo Vision, которая вначале может создавать определенные трудности в отладке.

В этой главе обсуждаются дополнительные средства, имеющиеся в Turbo Vision и увеличивающие эффективность использования этой библиотеки. Кроме того, приводятся практические рекомендации по разработке всей программы в целом и программированию отдельных ее частей.

23.1. КОНТРОЛЬ ЗА ДИНАМИЧЕСКОЙ ПАМЯТЬЮ

Как правило, объекты в Turbo Vision размещаются в куче. Это отвечает специфике диалоговых программ: на этапе разработки программист обычно не может учесть все возможные действия пользователя программы. Чтобы не накладывать неестественные ограничения на те или иные ее возможности, не следует злоупотреблять статическими определениями объектов, так как в этом случае программа не сможет гибко учитывать специфические требования пользователя.

В правильно разработанной программе контроль за доступной динамической памятью должен осуществляться перед каждым обращением к *New*, в противном случае нехватка памяти может вызвать аварийный останов программы и все подготовленные пользователем данные и промежуточные результаты будут безвозвратно потеряны.

В Turbo Vision имеются средства, упрощающие этот контроль: глобальная функция *LowMemory* будет возвращать *True*, если размер свободного участка кучи стал слишком мал (по умолчанию меньше 4 Кбайт). Таким образом, вместо того, чтобы контролировать кучу перед каждым обращением к *New*, можно обратиться к функции *LowMemory* перед началом размещения динамического объекта или сразу после того, как объект размещен в куче. Если *LowMemory* возвращает *True*, дальнейшая работа с кучей возможна только после ее очистки. Резервный участок кучи длиной в 4 Кбайт называется пулом надежности. Предполагается, что его размеры достаточны для размещения любого объекта Turbo Vision, поэтому обычно контроль с помощью *LowMemory* осуществляется сразу после процедуры динамического размещения нового видимого элемента.

В следующем примере создается простое диалоговое окно:

```
Uses Memory,...; {Функция LowMemory
                  определена в модуле Memory}
....
R.Assign(20,3,60,10);
D := New(Dialog, Init(R, 'Диалоговое окно'));
with D^ do
  begin
    R.Assign(2,2,32,3);
    Insert(New(PStaticText, Init(R, 'Сообщение-вопрос')));
    R.Assign(5,5,14,7);
    Insert(New(PButton, Init(R, '~Y~es (Да)', cmYes)));
    R.Assign(16,5,25,7);
    Insert(New(PButton, Init(R, '~N~o (Нет)', cmNO)));
  end;
if LowMemory then
  begin
    Dispose(D,Done); {Нет памяти: удаляем распределение}
    OutOfMemory;      {Сообщаем об этом}
    DoIt := False     {Признак ошибки}
  end
else
  DoIt := DeskTop^.ExecView(D)=cmYes;
```

Если Вы используете вызов *LowMemory* сразу после динамического размещения объекта, то в ходе самого размещения не должен произойти аварийный останов, связанный с нехваткой памяти. Таким образом, размер пула надежности должен быть достаточным для размещения всего объекта. Переменная *LowMemSize* задает размер пула надежности в параграфах (участках, длиной по 16 байт). По умолчанию она имеет значение $4096 \div 16 = 256$, т.е. размер пула надежности составляет 4 Кбайт.

На практике вместо прямого обращения к *LowMemory* чаще используется вызов метода *TProgram.ValidView (P: Pointer): Pointer*. Этот метод получает в качестве параметра обращения указатель *P* на динамический объект и осуществляет следующие действия:

- если *P = NIL*, метод возвращает *NIL*;
- если *LowMemory = True*, метод освобождает память, связанную с *P*, вызывает метод *TProgram.OutOfMemory* и возвращает *NIL*;
- если обращение к методу *TView.Valid (cmValid)* дает *False* (см. ниже), объект *P* удаляется из кучи и метод *ValidView* возвращает *NIL*;
- в противном случае считается, что размещение осуществлено успешно, и метод возвращает значение указателя *P*.

Метод *TProgram.ValidView* осуществляет стандартные действия по контролю надежности использования кучи. Обычно его используют перед тем, как поместить новый видимый элемент в группу, например:

```
DeskTop^.Insert(ValidView(New(TMyWindow, Init(...))));
```

Заметим, что нехватка памяти вызывает обращение к виртуальному методу *OutOfMemory*, предназначенному для выдачи сообщения о ненормальной ситуации. По умолчанию этот метод ничего не делает и просто возвращает управление вызывающей программе. Вы должны перекрыть его, если хотите сообщить пользователю о возникшей проблеме.

В ряде случаев может оказаться полезной глобальная функция **Function** *MemAlloc* (*Size: Word*): *Pointer*, которая осуществляет те же действия, что и *New* или *GetMem*, но в отличие от них не распределяет пул надежности. Функция возвращает указатель на выделенную область кучи или *NIL*, если в куче нет свободного блока нужного размера. Аналогичные действия осуществляет функция *MemAllocSeg*, отличающаяся от *MemAlloc* только тем, что выделяет память, выровненную на границу параграфа (на границу сегмента).

23.2. ОБРАБОТКА ОШИБОК ИНИЦИАЦИИ И МОДАЛЬНЫХ СОСТОЯНИЙ

Каждый видимый элемент наследует виртуальный метод *TView.Valid* (*Command: Word*): *Boolean*. С помощью этого метода решаются две задачи: если параметр обращения *Command* = *cmValid* = 0, метод должен проверить правильность инициации объекта и выдать *True*, если инициация прошла успешно; при обращении с параметром *Command* <> *cmValid* метод возвращает *True* только тогда, когда модальное состояние диалогового элемента можно завершить командой *Command*. По умолчанию метод *Valid* возвращает *True*. Вы должны перекрыть этот метод, если хотите автоматизировать контроль за инициацией объекта и/или за завершением работы модального элемента.

Поскольку метод *Valid* автоматически вызывается из метода *ValidView*, в нем нет необходимости контролировать правильность использования кучи — это делает *ValidView*. С другой стороны, в методе *Valid* можно проверить другие условия правильного функционирования объекта. Например, если в объекте используется дисковый файл, можно проверить существование этого файла.

Типичный метод *Valid* имеет такой вид:

```
Function TMyView.Valid(Command: Word): Boolean;
begin
    Valid := True;
    {Проверяем корректность инициации;}
    if Command = cmValid then
        if not CorrectlyInit then
            begin
                ReportErrorInit; {Сообщаем о некорректной инициации}
                Valid := False
            end
        else
            {Проверяем корректность завершения;}
            else if Command <> EnableCommand then
                begin
```

```

ReportErrorEnd      {Сообщаем о некорректном выходе}
Valid := False
end
end;

```

В этом фрагменте предполагается, что результат проверки правильности создания элемента возвращается в логической переменной *CorrectlyInit*, проверка корректности завершения работы модального элемента осуществляется сравнением команды завершения с ожидаемой командой *EnableCommand*, а сообщения об обнаруженных отклонениях от нормы выдаются процедурами *ReportErrorInit* и *ReportErrorEnd*. Заметим, что сообщения об ошибках инициации, не связанных с динамическим распределением объекта в куче, реализуются в методе *Valid*, в то время как сообщения об ошибках кучи — в методе *ValidView*.

Если видимый элемент — модальный, метод *Valid* перекрывается также для того, чтобы сообщить вызывающей программе о том, будет ли корректным в данном контексте завершение модального состояния командой *Command* или нет. Таким способом можно, например, перехватить выход из окна редактора в случае, если в нем остался несохраненный в файле текст. Разумеется, в подобной ситуации программа может не только выдать сообщение пользователю, но и предпринять необходимые действия для корректного завершения работы модального элемента. Метод *Valid* автоматически вызывается методом *Execute* модального элемента перед завершением работы.

Перекрытие методов *Valid* особенно полезно на этапе создания сложных программ: автоматический вызов проверки состояния используемого видимого элемента и выдача исчерпывающей диагностики могут дать неоценимую помощь программисту. Если Вы перекрываете методы *Valid*, Вы можете программировать, не заботясь о многочисленных проверках — методы сделают это за Вас.

23.3. ОТЛАДКА ПРОГРАММ

Если Вы пытались отлаживать какую-либо программу в Turbo Vision, Вы наверняка убедились, что трассировка (пошаговое проследивание логики работы) таких программ весьма неэффективна. Вызвано это двумя обстоятельствами. Во-первых, значительная часть библиотеки Turbo Vision скрыта от Вас: библиотека поставляется в TPU-файлах, проследивание работы которых которых невозможно. Во-вторых, в Turbo Vision используется принцип отделения логики создания видимых элементов от логики обработки связанных с ними событий: как только видимый элемент активизируется вызовом *Execute*, начинает работать его метод *HandleEvent*, который может породить целую цепочку непрослеживаемых трассировкой действий программы.

Ключом к решению проблемы отладки программ в Turbo Vision является расстановка точек контроля в наследуемых методах *HandleEvent*. Если программа не хочет открывать диалоговое окно или не реагирует на нажимаемую кнопку, следует прежде всего убедиться в том, что Ваши действия действительно порождают нужное событие.

Может случиться, что установленная контрольная точка не будет реагировать вообще или, наоборот, будет активизироваться слишком часто. Если точка не активизируется, это означает, что Ваш обработчик событий просто «не видит» событие. В этом случае необходимо убедиться в том, что поле *EventMask* видимого объекта содержит маску, позволяющую ему реагировать на событие нужного вида. Другой причиной «исчезновения» события может быть его перехват (и обработка) другим видимым элементом. Это может быть вызвано различными обстоятельствами. Например, Вы могли ошибочно связать две разные команды с одной константой или используете команду, которую использует также другой видимый элемент. Кроме того, обычно в наследуемых методах *HandleEvent* вызывается обработчик событий объекта-родителя, который может «украсть» событие у Вашего обработчика. В таких ситуациях бывает достаточно сделать вызов родительского метода после того, как событие будет обработано Вами.

Если контрольная точка активизируется слишком часто, значит Вы установили ее неправильно. Например, если Вы установили эту точку внутри метода *TGroup.Execute*, точка будет непрерывно активизироваться, т.к. значительная часть времени работы программы тратится на ожидание события. Если Вам все-таки требуется установить контрольную точку именно в этом месте, сделайте ее условной, чтобы она не реагировала на пустые или ненужные события.

Иногда запущенная программа «зависает», т.е. перестает реагировать на любые действия пользователя. Такие ошибки отлаживать труднее всего. Если программа «зависла», попытайтесь прежде всего локализовать то место, в котором это происходит. Для этого обычно используется расстановка контрольных точек в подозрительных местах программы. Следует помнить, что в Turbo Vision «зависания» связаны в основном с тремя видами ошибок:

- освобождается динамический объект, который входил в состав ранее освобожденной динамической группы;
- читаются данные из потока в ошибочно зарегистрированный объект (объект имеет неуникальный регистрационный номер);
- элемент коллекции ошибочно трактуется как элемент другого типа.

Ошибки первого вида встречаются наиболее часто. Например, прогон следующего невинного на первый взгляд варианта программы приводит к зависанию:

```
Uses Objects, Views;
```

```
var
```

```
  G1, G2: PGroup;
```

```
  R: TRect;
```

```
begin
```

```
  R.Assign(10, 5, 70, 20);
```

```
  G1 := New(PGroup, Init(R));
```

```
  R.Grow(-10, -3);
```

```
  G2 := New(PGroup, Init(R));
```

```
  G1^.Insert(G2);
```

```

Dispose(G1, Done);
Dispose(G2, Done) {Здесь программа "зависнет"!}
end.

```

Заметим, что перестановка операторов *Dispose* местами приводит к корректному варианту, т.к. метод *G1.Done* умеет контролировать освобождение своего подэлемента *G2* и не освобождает его вторично. Во всех случаях оператор *Dispose (G2, Done)* излишен: освобождение группы вызывает автоматическое освобождение всех ее подэлементов.

Поскольку динамическая память используется в Turbo Vision очень интенсивно, полезно предусмотреть в отладочном варианте программы визуализацию ее размера. Для этого можно использовать такой объект *THeapView*:

```

Unit HeapView;
Interface
Uses Dialogs, Objects;
type
  PHeapView = ^THeapView;
  THeapView = object(TStaticText)
    Constructor Init(var R: TRect);
    Procedure Update;
  end;
Implementation
Constructor THeapView.Init;
var
  S: String;
begin
  Str(MemAvail, S);
  Inherited Init(R, #3+S)
end;
Procedure THeapView.Update;
var
  S: String;
begin
  Str(MemAvail, S);
  DisposeStr(Text);
  Text := NewStr(#3+S);
  Draw
end;
end.

```

Например, в следующей программе показан способ включения контрольного окна, создаваемого в этом объекте, в верхний правый угол экрана:

```

Uses Objects, Views, App, HeapView;
var
  H: PHeapView;           {Окно для MemAvail}

```

```

W: PWindow;
G: PGroup;
R: TRect;
P: TApplication;           {Стандартная программа}
begin
  P.Init;
  R.Assign(70,0,80,1); {Верхний правый угол}
  New(H, Init(R));      {Создаем окно контроля}
  P.Insert(H);           {Помещаем его на экран}
  ReadLn;                {Пауза - показываем начальный размер кучи}
  R.Assign(10,5,70,20);
  W := New(PWindow, Init(R, '', 0)); {Создаем окно}
  R.Assign(5,3,55,12);
  G := New(PGroup, Init(R));
  W^.Insert(G);          {Вставляем в окно группу}
  DeskTop^.Insert(W);    {Выводим на экран}
  H^.Update;             {Обновляем окно контроля}
  ReadLn;                {Пауза - размер кучи перед освобождением}
  Dispose(W, Done);      {Освобождаем окно и группу}
  H^.Update;             {Обновляем окно контроля}
  ReadLn;                {Пауза - размер после освобождения}
  P.Done
end.

```

Для получения текущего значения общего размера кучи используется вызов метода *THeapView.Update* в нужных местах программы. Вы можете автоматизировать обновление окна контроля, если включите вызов *Update* в перекрываемый метод *TProgram.Idle*. В следующем варианте показан способ отображения *MemAvail* в фоновом режиме. Кроме того, в программе иллюстрируется возможное использование функции *MessageBox*.

```

{$X+} {Используется расширенный синтаксис
вызова функции MessageBox}

```

```

Uses Objects, Views, App, HeapView, MsgBox;

```

```

type

```

```

  MyApp = object (TApplication)

```

```

    Procedure Idle; Virtual;

```

```

  end;

```

```

var

```

```

  H: PHeapView;

```

```

Procedure MyApp.Idle;

```

```

begin

```

```

  H^.Update

```

```

end;

```

```

var

```

```

  W: PWindow;

```

```

  G: PGroup;

```

```

  R: TRect;

```

```

  P: MyApp;

```

begin

```
P.Init;
R.Assign(70,0,80,1);
New(H, Init(R));
P.Insert(H);
MessageBox(#3'Размер кучи до размещения', NIL, 0);
R.Assign(10,5,70,20);
W := New(PWindow, Init(R, '', 0));
R.Assign(5,3,55,12);
G := New(PGroup, Init(R));
W^.Insert(G);
DeskTop^.Insert(W);
MessageBox(#3'Размер кучи после размещения', NIL, 0);
Dispose(W, Done);
MessageBox(#3'Размер кучи после освобождения', NIL, 0);
P.Done
```

end.

Константа #3 вставляется в начало строки сообщения в том случае, когда требуется центрировать эту строку (расположить ее симметрично относительно границ окна сообщения).

23.4. ИСПОЛЬЗОВАНИЕ ОВЕРЛЕЯ

Модули Turbo Vision разработаны с учетом возможного использования их в оверлейных программах. Все они могут быть оверлейными за исключением модуля *Drivers*, который содержит процедуры обработки прерываний и другой системный интерфейс низкого уровня.

При разработке оверлейных программ старайтесь спроектировать логические последовательности вызовов тех или иных модулей так, чтобы по возможности уменьшить свопинг (динамический обмен оверлеев). Поскольку программы Turbo Vision рассчитаны на интенсивное использование диалога с пользователем, размещайте (если это возможно) все процедуры, связанные с некоторой точкой диалога, в том же модуле, в котором создается и выполняется соответствующий модальный элемент. Например, используемые в программе диалоговые окна, как правило, порождаются от *TDialog*, а диалоговые элементы этих окон — от *TInputLine* и *TListViewer*. Если Вы поместите все три порожденных типа в один модуль, Ваша программа будет выполняться быстрее, так как взаимосвязанные вызовы объектов не будут приводить к свопингу оверлеев.

Заметим, что размеры всех основных оверлейных модулей — *App*, *Objects*, *Views*, *Menus* приблизительно одинаковы и составляют около 50 Кбайт. С учетом Ваших собственных объектов, обеспечивающих интенсивное взаимодействие с пользователем и порожденных от *TWindow* или *TDialog*, типичный размер оверлейного буфера составит не менее 64 Кбайт. Если Вы хотите минимизировать потери времени на свопинг и в то же время создать достаточно компактную программу, Вам придется поэкспериментировать с размером оверлейного буфера и/или испытательной зоны. Вы можете также возложить на пользователя Вашей программы

ответственность за выбор размера оверлейного буфера, предусмотрев соответствующий параметр в строке вызова программы.

В следующем примере показан возможный способ инициации оверлея.

```
{ $F+,O+,S- }
{ $M 8192,65536,655360 }
Uses Overlay, Drivers, Memory, Objects, Views, Menus,
Dialogs, istList, StdDlg, App;
{ Объявляем оверлейные модули: }
{ $O App      }
{ $O Dialogs  }
{ $O HistList }
{ $O Memory   }
{ $O Menus    }
{ $O Objects  }
{ $O StdDlg   }
{ $O Views    }
const
  OvrBufDisk  = 96 * 1024; { Размер буфера без EMS-памяти }
  OvrBufEMS   = 72 * 1024; { Размер буфера при
                             использовании EMS-памяти }

type
  TMyApp = object (TApplication)
    Constructor Init;
    Destructor Done; Virtual;
    .....
  end; { TMyApp }

Procedure InitOverlays;
var
  FileName: String [79];
begin
  FileName := ParamStr(0);
  OvrInit(FileName);
  if OvrResult <> 0 then
    begin
      PrintStr('Невозможно открыть оверлейный файл ', FileName);
      Halt;
    end;
  OvrInitEMS;
  if OvrResult = 0 then
    OvrSetBuf(OvrBufEMS)
  else
    begin
      OvrSetBuf(OvrBufDisk);
      OvrSetRetry(OvrBufDisk div 2);
    end
  end; { InitOverlays }
Constructor TMyApp.Init;
```

```

begin
  InitOverlays;
  TApplication.Init;
  .....
  .....
end; {TMyApp.Init}
Destructor TMyApp.Done;
begin
  .....
  .....
  TApplication.Done;
end; {TMyApp.Done}
var
  MyApp: TMyApp;
begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
end.

```

В этой программе используется механизм размещения оверлеев в исполняемом *EXE*-файле. Для этого после компиляции программы используйте команду ДОО

```
copy/b MyProg.exe+MyProg.ovr MyProg.exe
```

Чтобы определить маршрут поиска *EXE*-файла, в процедуре *InitOverlays* проверяется параметр вызова с номером 0. Заметим, что в этот параметр автоматически помещается маршрут доступа к загруженной программе только в том случае, когда Ваш ПК работает под управлением *MS-DOS* версии 3.0 и выше.

Обратите внимание: размер оверлейного буфера можно сделать меньшим, если программа обнаружит доступную *EMS*-память, ведь в этом случае потери времени на свопинг будут минимальными. Разумеется, инициация оверлея осуществляется до обращения к *TApplication.Init*, т.к. модуль *App*, в котором находится этот метод, сделан в программе оверлейным.

23.5. ПОРЯДОК ВЫЗОВА НАСЛЕДУЕМОГО МЕТОДА

Большая часть объектов Turbo Vision спроектирована в расчете на их дальнейшее перекрытие в прикладных программах. Типичным примером такого рода объектов является *TView*, метод *Draw* которого создает на экране пустой прямоугольник и, следовательно, не может отображать никакой полезной информации. Поскольку все видимые элементы порождены от *TView*, Вам необходимо перекрыть метод *Draw* в собственном объекте — потомке. Более того, поскольку *TView.Draw* не делает никакой полезной работы, его не нужно вызывать в перекрытом методе. Однако полностью перекрываемые методы, подобные *TView.Draw*, скорее исключение из общего правила. Обычно в перекрытом методе вызывается со —

ответствующий метод, наследуемый от родителя, т.к. в нем реализуются некоторые необходимые для потомка действия. В такого рода ситуациях важна последовательность вызова наследуемого метода: вызывать ли его до реализации специфичных действий или после? Ниже приводятся практические рекомендации на этот счет.

23.5.1. Конструктор

Вызывайте наследуемый метод до реализации дополнительных действий:

```
Procedure MyObject.Init(.....);
begin
  {Вызов наследуемого конструктора Init}
  {Реализация дополнительных действий}
end;
```

Такая последовательность необходима по той простой причине, что вызов наследуемого конструктора приводит к обнулению всех дополнительных полей объекта *MyObject*. Если, например, Вы используете следующий фрагмент программы:

```
type
  MyObject = object (TWindow)
    Value: Word;
    Ok   : Boolean;
    Constructor Init(var Bounds: TRect; ATitle: TTitleStr;
      AValue: Word; AOk: Boolean);
end;
Constructor MyObject.Init;
begin
  Inherited Init(Bounds, ATitle, wnNoNumber);
  Value := 16;
  Ok := True;
end;
```

то дополнительные поля *Value* и *Ok* получают нужные значения 16 и *True*. Однако, если обращение *TWindow.Init (Bounds, ATitle, wnNoNumber)*; поставить после оператора *Ok := True*, в них будут помещены значения 0 и *False*. Из этого правила существует одно исключение, связанное с загрузкой коллекции из потока конструктором *Load*. Дело в том, что в наследуемом методе *TCollection.Load* реализуется следующий цикл:

```
Constructor TCollection.Load (var S: TStream);
begin
  .....
  for I := 0 to Count - 1 do
    AtPut(I, GetItem(S));
end;
```

Если элементами коллекции являются произвольные наборы двоичных данных (не объекты), Вам потребуется перед чтением очередного элемента

сначала получить из потока его длину. Следующий пример иллюстрирует сказанное.

type

```
PDataCollection = ^TDataCollection;
TDataCollection = object (TStringCollection)
  ItemSize: Word;
  Constructor Load(var S: TStream);
  Function GetItem(var S: TStream): Pointer; Virtual;
  .....
```

end;

```
Constructor TDataCollection.Load(var S: TStream);
```

begin

```
  S.Read(ItemSize, SizeOf(ItemSize));
```

```
  Inherited Load(S);
```

end;

```
Function TDataCollection.GetItem(var S: TStream): Pointer;
```

var

```
  Item: Pointer;
```

begin

```
  GetMem(Item, ItemSize);
```

```
  S.Read(Item^, ItemSize);
```

```
  GetItem := Item;
```

end;

В этом примере конструктор *Load* сначала загружает из потока поле *ItemSize*, содержащее длину читаемого элемента. Затем вызывается конструктор *TCollection.Load*, в котором осуществляется вызов *GetItem*. Новый *GetItem* использует поле *ItemSize*, чтобы определить размер читаемых данных, и резервирует нужный буфер в динамической памяти. Разумеется, запись полиморфных коллекций в поток должна происходить в том же порядке, т.е. сначала записывается длина очередного элемента, а уже потом — его данные.

23.5.2. Деструктор

Вызывайте наследуемый метод после реализации дополнительных действий:

```
Procedure MyObject.Done;
```

begin

```
  {Реализация дополнительных действий}
```

```
  {Вызов наследуемого деструктора Done}
```

end;

Работа деструктора проходит в обратном порядке по отношению к конструктору. Вначале Вы должны освободить всю дополнительно распределенную динамическую память, а уже затем вызвать наследуемый деструктор, чтобы уничтожить весь объект.

23.5.3. Другие методы

Порядок вызова наследуемого метода зависит от конкретного алгоритма. В большинстве случаев наследуемый метод вызывается первым, но могут использоваться и другие последовательности. Особое значение имеет вызов наследуемого обработчика событий *HandleEvent*. В самом общем виде структура нового обработчика будет такой:

```
Procedure MyObject.HandleEvent(var Event: TEvent);  
begin  
  {Изменение наследуемых свойств}  
  {Вызов наследуемого обработчика}  
  {Добавление новых свойств}  
end;
```

Таким образом, вначале Вы должны запрограммировать те действия, которые изменяют стандартное поведение перекрытого обработчика, затем вызвать его и, наконец, осуществить новую обработку событий. Разумеется, любая из этих трех частей может отсутствовать. Например, стандартный обработчик *TDialog.HandleEvent* лишь расширяет свойства наследуемого метода *TWindow.HandleEvent*, добавляя в него обработку событий от клавиатуры и событий — команд:

```
Procedure TDialog.HandleEvent(var Event: TEvent);  
begin  
  Inherited HandleEvent(Event);  
  case Event.What of  
    evKeyDown:  
      .....  
    evCommand:  
      .....  
  end;  
end;
```

Этот обработчик перехватывает все события от клавиатуры и мыши, в том числе и нажатие на клавишу *Tab*. Если Вы хотите обработать событие от клавиши *Tab* особым способом, Вы должны перехватить это событие до вызова стандартного обработчика. Например:

```
Procedure TNoTabsDialog.HandleEvent(var Event: TEvent);  
begin  
  if (Event.What = evKeyDown) then  
    if (Event.KeyCode = kbTab) or  
      (Event.KeyCode = kbShiftTab) then  
      ClearEvent(Event);  
  Inherited HandleEvent(Event);  
end;
```

23.6. ПРИМЕРЫ ПРОГРАММНЫХ РЕАЛИЗАЦИЙ

В этом разделе приводятся примеры программных реализаций некоторых типичных задач, решаемых с помощью Turbo Vision. Эти примеры не имеют сколько-нибудь серьезного прикладного назначения, они просто иллюстрируют возможные приемы программирования. Поскольку большинство видимых объектов используется в тесной взаимосвязи, примеры иллюстрируют программирование сразу нескольких объектов.

23.6.1. Строка статуса

В следующей программе создается строка статуса, содержание которой зависит от установленного контекста подсказки (определяется значением поля *TProgram.HelpCtx*). В зависимости от действий пользователя эта строка будет содержать текст

Esc Выход F1 Сменить контекст на 1
Однажды в студеную, зимнюю пору

либо

ESC Выход F2 Сменить контекст на 0
Я из лесу вышел. Был сильный мороз...

Переключение строки осуществляется клавишами *F1* и *F2*, для выхода из программы используется клавиша *Esc*.

```
Uses Objects, App, Menus, Drivers, Views;
type
  PMyStatusLine = ^TMyStatusLine;
  TMyStatusLine = object (TStatusLine)
    Function Hint(Cntx: Word): String; Virtual;
  end;
  MyApp = object (TApplication)
    StatLine: PMyStatusLine;
    Constructor Init;
    Procedure InitStatusLine; Virtual;
    Procedure HandleEvent(var Event: Tevent); Virtual;
  end;
const
  cmCntx1 = 200;
  cmCntx2 = 201;
  {-----}
Constructor MyApp.Init;
begin
  Inherited Init;
  Insert(StatLine) {Использовать нестандартную строку статуса}
end(MyApp.Init);
{-----}
Procedure MyApp.InitStatusLine;
{Инициация нестандартного поля MyApp.StatLine}
```

```
var
  R: Trect;
begin
  GetExtent(R);
  R.A.Y := pred(R.B.Y);
  StatLine := New(PMyStatusLine, Init(R,
    NewStatusDef(0,0, {Первый вариант строки}
    NewStatusKey('~Esc~ Выход', kbEsc, cmQuit,
    NewStatusKey('~F1~ Сменить контекст на 1',
      kbF1, cmCntx1, NIL)),
    NewStatusDef(1,1, {Второй вариант строки}
    NewStatusKey('~Esc~ Выход', kbEsc, cmQuit,
    NewStatusKey('~F2~ Сменить контекст на 0',
      kbF2, cmCntx2, NIL)),
    NIL))));
end{MyApp.InitStatusLine};
{-----}
Procedure MyApp.HandleEvent;
{Переключение контекста и обновление строки статуса}
begin
  Inherited HandleEvent(Event);
  case Event.Command of
    cmCntx1: HelpCtx := 1;
    cmCntx2: HelpCtx := 0;
  else
    ClearEvent(Event);
  end;
  if Event.What <> evNothing then
    begin
      StatLine^.Update;
      ClearEvent(Event)
    end
  end {MyApp.HandleEvent};
{-----}
Function TMyStatusLine.Hint(Cntx: Word): String;
{Переключение поля подсказки}
const
  Prompt: array [0..1] of String = (
    'Однажды в студеную, зимнюю пору',
    'Я из лесу вышел. Был сильный мороз...');
begin
  Hint := Prompt[Cntx]
end {TMyStatusLine.Hint};
{-----}
var
  P: MyApp;
begin
```

```

P.Init;
P.Run;
P.Done
end.

```

23.6.2. Меню

Стандартная программа содержит поле *MenuBar* типа *TMenuBar*. По умолчанию метод *TApplication.InitMenuBar* устанавливает это поле в *NIL*, что означает отказ от меню. Если Вы хотите использовать меню в Вашей программе, необходимо перекрыть этот метод. В следующем примере создается двухуровневое меню, показанное на рис.23.1.



Рис.23.1. Двухуровневое меню

Опцию «Подменю...» можно выбрать следующими способами:

- нажатием клавиш *F10* — <смещение указателя> — *Enter*;
- командой *Alt-S*;
- отметкой мышью.

Опции «Первый выбор» и «Второй выбор» можно выбрать клавишами *F1* и *F2* без разворачивания подменю. После разворачивания подменю можно использовать те же клавиши, а также использовать клавиши с цифрами 1 и 2, отметку мышью или смещение указателя к нужной опции и *Enter*. Опция «Третий выбор» доступна только после разворачивания подменю. Выбор каждой из этих трех опций приводит к появлению на экране окна с сообщением. Кроме того, опция «Третий выбор» попеременно запрещает или разрешает действие команд *cm1*, *cm2* и *cmQuit*.

```

{$X+}
Uses Objects,App,Menus,Drivers,Views,MsgBox;
type
  MyApp = object (TApplication)
    Procedure InitMenuBar; Virtual;
    Procedure HandleEvent(var Event: TEvent); Virtual;
  end;
const
  cm1 = 201;
  cm2 = 202;

```



```

    cm3 = 203;
    {-----}
Procedure MyApp.InitMenuBar;
var
    R: TRect;
begin
    GetExtent(R);
    R.B.Y := succ(R.A.Y);
    MenuBar := New(PMenuBar, Init(R,
    NewMenu(                                {Главная полоса меню}
    NewSubMenu(                             {Первый элемент главного меню}
    '~S~ Подменю...',hcNoContext,
    NewMenu(                                {Определить выпадающее подменю}
    NewItem('~1~ Первый выбор','F1',kbF1,cm1,0,
    NewItem('~2~ Второй выбор','F2',kbF2,cm2,0,
    NewLine(                                {Определить разделяющую линию}
    NewItem('~3~ Третий выбор','',0,cm3,0,
    NIL))))),
    NewItem(                                {Второй элемент главного меню}
    '~ESC~ Выход','~ESC~',kbEsc,cmQuit,0,
    NIL)))));
end {MyApp.InitMenuBar};
    {-----}
Procedure MyApp.HandleEvent;
const
    Flag: Boolean = True;
    cms = [cm1,cm2,cmQuit];
begin
    Inherited HandleEvent(Event);
    case Event.Command of
        cm1: MessageBox(#3'Первый выбор', NIL,0);
        cm2: MessageBox(#3'Второй выбор', NIL,0);
        cm3:
            begin
                MessageBox(#3'Третий выбор', NIL,0);
                if Flag then
                    DisableCommands(cms)
                else
                    EnableCommands(cms);
                Flag := not Flag
            end
    end
end {MyApp.HandleEvent};
    {-----}
var
    P: MyApp;
begin

```

```

P.Init;
P.Run;
P.Done
end.

```

23.6.3. Диалоговое окно

На рис.23.2 показан вид диалогового окна, которое создается и используется в рассматриваемом ниже примере.

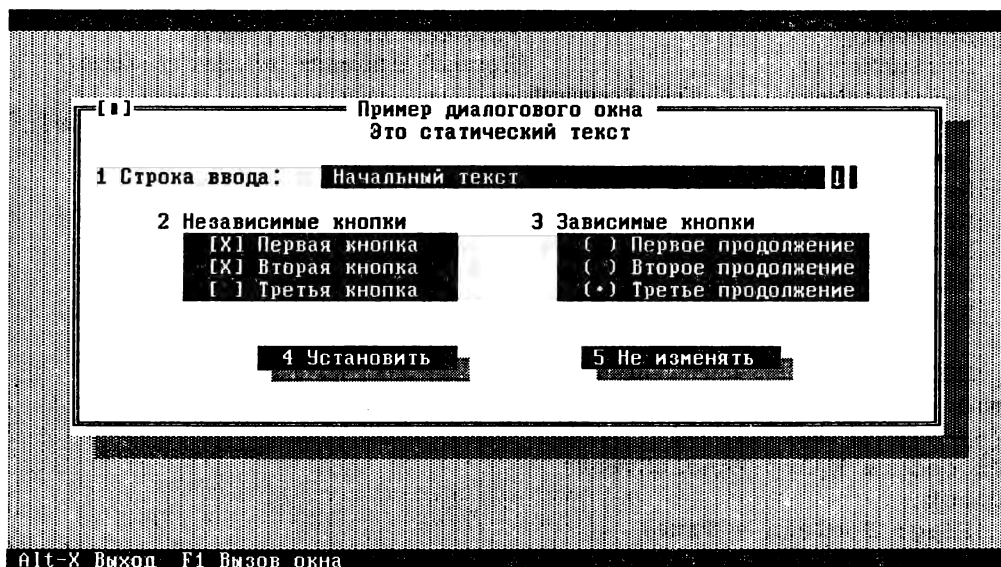


Рис.23.2. Диалоговое окно со строкой ввода и кнопками

Если диалог завершен командой «Установить», на экране разворачивается окно, в котором сообщаются полученные из диалогового окна параметры — текст из строки ввода и настройка кнопок. Если диалог завершен командой «Не изменять», окно сообщений содержит строку

Команда 'Не изменять',

если диалог завершен по клавише *Esc*, на экран не выводится никаких сообщений.

Пример иллюстрирует использование строки ввода, зависимых и независимых кнопок и нестандартных команд.

Для задания начальных параметров и чтения параметров, установленных в результате диалога, используется поле *TDialog.Data*. Это поле содержит данные, используемые в ходе диалога, в виде записи, поля и тип которой строго соответствуют порядку и типу вставляемых в окно терминальных элементов. В нашем примере (см. текст программы) первым в окно вставляется статический текст, однако этот терминальный элемент не имеет параметров, которые можно было бы изменить в ходе диалога, поэтому в записи *Data* ему не отводится место. Второй по счету в окно

вставляется строка ввода *TInputLine*. Этот объект характеризуется длиной *L* строки, указываемой вторым параметром обращения к *TInputLine.Init*, поэтому для него в *Data* выделяется поле, длиной *L+1* байт. Каждому кластеру с кнопками выделяется поле *WORD*, что дает возможность задать в кластере до 16 независимых кнопок и до 65536 зависимых: каждая независимая кнопка связана с соответствующим разрядом 16-битного поля (первая кнопка — с младшим разрядом), а каждой зависимой кнопке соответствует свое число (первой кнопке соответствует число 0, второй — 1 и т.д.). Установка данных в поле *TDIALOG.Data* осуществляется методом *TDIALOG.SetData*, получить данные после диалога можно с помощью метода *TDIALOG.GetData*.

{ \$X+ }

Uses Objects, App, Drivers, Dialogs, Menus, Views, MsgBox;

type

MyApp = **object** (TApplication)

Procedure InitStatusLine; **Virtual**;

Procedure HandleEvent(**var** Event: TEvent); **Virtual**;

Procedure GetDialog;

end;

PMyDialog = ^TMyDialog;

TMyDialog = **object** (TDIALOG)

Procedure HandleEvent(**var** Event: TEvent); **Virtual**;

end;

const

 cm0 = 200;

 cm1 = 201;

 cm2 = 202;

{-----}

Procedure MyApp.InitStatusLine;

{Создает строку статуса}

var

 R: TRect;

begin

 GetExtent(R);

 R.A.Y := pred(R.B.Y);

 StatusLine := New(PStatusLine, Init(R,

 NewStatusDef(0, \$FFFF,

 NewStatusKey('~Alt-X~ Выход', kbAltX, cmQuit,

 NewStatusKey('~F1~ Вызов окна', kbF1, cm0, NIL)),

 NIL));

end {MyApp.InitStatusLine};

{-----}

Procedure MyApp.HandleEvent;

{Обрабатывает нестандартную команду cm0}

begin

Inherited HandleEvent(Event);

case Event.Command **of**

```

    cm0: GetDialog
  else
    ClearEvent(Event)
  end
end {MyApp.HandleEvent};
{-----}
Procedure MyApp.GetDialog;
{Создает и использует диалоговое окно}
var
  R: TRect;                {Координаты элементов}
  D: PMyDialog;            {Диалоговое окно}
  I: PInputLine;           {Строка ввода}
  RB: PRadioButtons;       {Зависимые кнопки}
  CB: PCheckBoxes;         {Независимые кнопки}
  s : String;              {Для вывода сообщения}
const
  L = 120;                 {Длина строки ввода}
type
  TDialogData = record    {Параметры диалогового окна}
    I_Data: String [L];   {Текст в строке ввода}
    CB_Data: Word;         {Независимые кнопки}
    RB_Data: Word;         {Зависимые кнопки}
  end;
const
  st: array [0..2] of String = ('Первое', 'Второе', 'Третье');
  Data : TDialogData = (    {Начальные параметры диалога}
    I_Data : 'Начальный текст';
    CB_Data: 3;              {1-я и 2-я кнопка}
    RB_Data: 2);            {3-е продолжение}
begin
  R.Assign(5,3,75,18);      {Координаты диалогового окна}
  D := New(PMyDialog, Init(R, 'Пример диалогового окна'));
  with D^ do
    begin
      R.Assign(1,1,69,3);
      Insert(New(PStaticText, {Вставляем статический текст}
        Init(R, #3'Это статический текст')));
      R.Assign(20,3,60,4);
      I := New(PInputLine, Init(R, L));
      Insert(I);              {Вставляем строку ввода}
      R.Assign(1,3,20,4);
      Insert(New(PLabel,      {Вставляем метку строки ввода}
        Init(R, '~1~ Строка ввода:', I)));
      R.Assign(60,3,62,4);
      Insert(New(PHistory,    {Вставляем список ввода}
        Init(R, I, 0)));
      R.Assign(10,6,30,9);
      CB := New(PCheckBoxes, Init(R,

```

```

NewSItem('Первая кнопка',
NewSItem('Вторая кнопка',
NewSItem('Третья кнопка', NIL))));
Insert(CB); {Вставляем независимые кнопки}
R.Assign(6,5,30,6);
Insert(New(PLabel, {Вставляем метку кнопок}
Init(R,'~2~ Независимые кнопки',CB)));
R.Assign(40,6,63,9);
RB := New(PRadioButtons, Init(R,
NewSItem('Первое продолжение',
NewSItem('Второе продолжение',
NewSItem('Третье продолжение', NIL))));
Insert(RB); {Вставляем зависимые кнопки}
R.Assign(36,5,63,6);
Insert(New(PLabel, {Вставляем метку кнопок}
Init(R,'~3~ Зависимые кнопки',RB)));
R.Assign(14,11,32,13);
Insert(New(PButton, {Вставляем кнопку "Установить"}
Init(R,'~4~ Установить',cm1,bfNormal)));
R.Assign(40,11,58,13);
Insert(New(PButton, {Вставляем кнопку "Не изменять"}
Init(R,'~5~ Не изменять',cm2,bfNormal)));
SetData(Data) {Устанавливаем начальные значения}
end;
{Помещаем окно на экран и получаем команду завершения}
case DeskTop^.ExecView(D) of
cm1:
begin {Была команда "Установить":}
D^.GetData(Data); {Получаем новые значения}
with Data do
begin {Готовим сообщение}
s := #3'Параметры диалогового окна: '+
#13'Текст :'+I_Data+#13'Кнопки: ';
if CB_Data and 1 <> 0 then
s := s+' Первая';
if CB_Data and 2 <> 0 then
s := s+' Вторая';
if CB_Data and 4 <> 0 then
s := s+' Третья';
s :=s+#13'Продолжение: '+st[RB_Data]
end
end;
cm2: s := #3'Команда "Не изменять"';
else
s := '';
end; {case}
if s <> '' then

```

```

    MessageBox(s, NIL, 0)
end {MyApp.GetDialog};
{-----}
Procedure TMyDialog.HandleEvent;
{Обрабатывает нестандартные команды cm1 и cm2}
begin
    Inherited HandleEvent(Event);
    if Event.What = evCommand then
        case Event.Command of
            cm1: EndModal(cm1);      {Завершить с командой cm1}
            cm2: EndModal(cm2);      {Завершить с командой cm2}
        end;
        ClearEvent(Event)           {Очистить другие события}
    end {TMyDialog.HandleEvent};
{-----}
var
    P: MyApp;
begin
    P.Init;
    P.Run;
    P.Done
end.

```

Для использования нестандартных команд *cm0*, *cm1* и *cm2* перекрываются обработчики событий строки статуса и диалогового окна. Чтобы завершить диалог с выдачей нестандартной команды, в обработчике событий окна вызывается метод *EndModal*, в результате чего метод *ExecView* возвращает значение соответствующей команды. Заметим, что стандартная для Turbo Vision команда *cmCancel* (закрыть окно по клавише *Esc*) обрабатывается стандартным методом *TDialog.HandleEvent*.

23.6.4. Окно с текстом

В следующей программе на экране создается несколько окон, содержащих один и тот же текст — текст программы (см. рис.23.3).

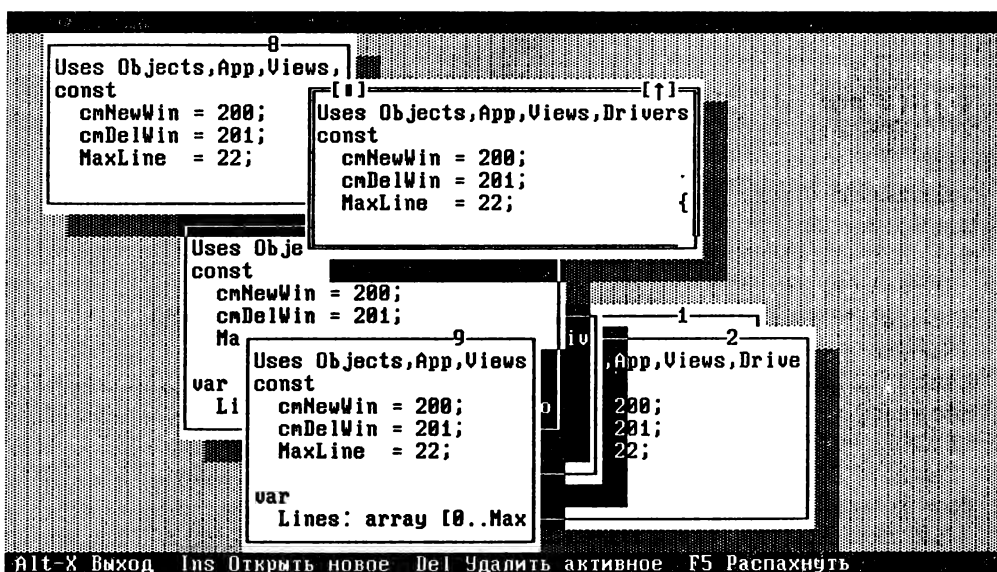


Рис.23.3. Окна с текстом программы

Каждое новое окно открывается с помощью клавиши *Ins*. Активное окно можно удалить клавишей *Del* или распахнуть на весь экран клавишей *F5*. С помощью мыши Вы можете перемещать активное окно по экрану и/или изменять его размеры.

```
Uses Objects,App,Views,Drivers,Menus;
const
  cmNewWin = 200;
  cmDelWin = 201;
  MaxLine = 22; {Количество текстовых строк}
var
  Lines: array [0..MaxLine] of String [80];
type
  MyApp = object (TApplication)
    WinNo : Word;
    Constructor Init;
    Procedure InitStatusLine; Virtual;
    Procedure HandleEvent(var Event: Tevent); Virtual;
    Procedure NewWindow;
  end;
  PInterior = ^TInterior;
  TInterior = object (TView)
    Constructor Init(R: TRect);
    Procedure Draw; Virtual;
  end;
{-----}
Constructor MyApp.Init;
{Открывает и читает файл с текстом программы}
```

```

var
  f: text;
  s: String;
  k: Integer;
begin
  Inherited Init;
  WinNo := 0; {ГОТОВИМ номер окна}
  for K := 0 to MaxLine do
    Lines[k] := ''; {ГОТОВИМ массив строк}
  s := copy(ParamStr(0),1,pos('.',ParamStr(0))+'PAS';
  {$I-}
  Assign(f,s);
  Reset(f);
  if IOResult <> 0 then
    exit; {Файл нельзя открыть}
  for k := 0 to MaxLine do
    if not EOF(f) then ReadLn(f,Lines[k]);
  Close(f)
  {$I+}
end {MyApp.Init};
{-----}
Procedure MyApp.InitStatusLine;
var
  R: TRect;
begin
  GetExtent(R);
  R.A.Y := pred(R.B.Y);
  StatusLine := New(PStatusLine, Init R,
  NewStatusDef(0,$FFFF,
  NewStatusKey('~Alt-X~ Выход',kbAltX,cmQuit,
  NewStatusKey('~Ins~ Открыть новое',kbIns,cmNewWin,
  NewStatusKey('~Del~ Удалить активное',kbDel,cmClose,
  NewStatusKey('~F5~ Распахнуть',kbF5,cmZoom, NIL)) ),
  NIL)))
end {MyApp.InitStatusLine};
{-----}
Procedure MyApp.HandleEvent;
{Обработка нестандартных команд cmNewWin, cmDelWin}
begin
  Inherited HandleEvent(Event);
  case Event.Command of
    cmNewWin:
      begin
        ClearEvent(Event);
        NewWindow;
      end;
    cmDelWin: Event.Command := cmClose;
  end;
end;

```



```

end;
ClearEvent(Event)
end {MyApp.HandleEvent};
{-----}
Procedure MyApp.NewWindow;
{Открывает новое окно}
var
  R: TRect;
  W: PWindow;
begin
  Inc(WinNo); {Номер окна}
  {Задаем случайные размеры и положение окна:}
  R.Assign(0,0,24+Random(10),7+Random(5));
  R.Move(Random(80-R.B.X),Random(24-R.B.Y));
  W := New(PWindow, Init(R,'',WinNo));
  W^.GetClipRect(R); {Получаем в R границы окна}
  R.Grow(-1,-1); {Размер внутренней части окна}
  {Инициализируем просмотр текста:}
  W^.Insert(New(PInterior, Init(R)));
  DeskTop^.Insert(W); {Помещаем окно на экран}
end {MyApp.NewWindow};
{-----}
Constructor TInterior.Init;
{Инициализация просмотра текста во внутренней части окна}
begin
  Inherited Init(R);
  GrowMode := gfGrowHiX+gfGrowHiY
end {TInterior.Init};
{-----}
Procedure TInterior.Draw;
{Вывод текста в окне}
var
  k: Integer;
  B: TDrawBuffer;
begin
  for k := 0 to pred(Size.Y) do
    begin
      MoveChar(B, ' ',GetColor(1),Size.X);
      MoveStr(B, copy(Lines[k],1,Size.X),GetColor(1));
      WriteLine(0,k,Size.X,1,B)
    end
  end {TInterior.Draw};
{-----}
var
  P: MyApp;
begin
  P.Init;
  P.Run;

```

```
P.Done
end.
```

В программе объявляется тип *TInterior*, предназначенный для создания изображения во внутренней части окон. Его метод *Init* определяет способ связи объекта *TInterior* со стандартным объектом *TWindow*: оператор

```
GrowMode := gfGrowHiX+gfGrowHiY
```

задает автоматическое изменение размеров объекта *TInterior* при изменении размеров окна так, чтобы вся внутренняя часть окна была всегда заполнена текстом. Метод *TInterior.Draw* заполняет внутреннюю часть окон текстовыми строками, которые в ходе выполнения конструктора *TMyApp.Init* предварительно считываются из файла с исходным текстом программы в глобальный массив *Lines*. Для вывода текста сначала с помощью метода *MoveChar* буферная переменная *B* типа *TDRAWBuffer* заполняется пробелами, затем методом *MoveStr* в нее копируется нужный текст, а с помощью *WriteLine* содержимое переменной *B* помещается в видеопамять. Такая последовательность действий стандартна для вывода текстовых сообщений в Turbo Vision. Заметим, что функция *GetColor* (1) возвращает номер элемента палитры, связанный с обычным текстом; для выделения тестовых строк можно использовать вызов *GetColor* (2).

23.6.5. Окно со скроллером

Скроллером называется специальное окно, обеспечивающее просмотр (скроллинг) текста. Типичный скроллер — это окно редактора интегрированной среды системы Турбо Паскаля; его поведение Вам, очевидно, хорошо знакомо. Средства Turbo Vision обеспечивают стандартные функции скроллера для окна, создаваемого в приводимой ниже программе. В частности, это окно (см. рис.23.4) управляется мышью, реагирует на клавиши смещения курсора, оно может изменять размеры и свое положение на экране, его можно «распахнуть» на весь экран.

```
Uses Objects, App, Drivers, Menus, Views;
```

```
var
```

```
  Lines: PCollection; {Коллекция для хранения текстовых строк}
type
```

```
  TMyApp = object (TApplication)
```

```
    Procedure Run; Virtual;
```

```
  end;
```

```
  PInterior = ^TInterior;
```

```
  TInterior = object (TScroller)
```

```
    Constructor Init(R: TRect; SX,SY: PScrollBar);
```

```
    Procedure Draw; Virtual;
```

```
  end;
```

```

[!] Просмотр файла E:\NEWBOOK2\23.PAS [!]
Uses Objects,App,Drivers,Menus,Views,int5;
var
  Lines: PCollection;      {Коллекция для хранения текстовых строк}

type
  TMyApp = object(TApplication)
    Procedure Run; virtual;
  end;

  PInterior = ^TInterior;
  TInterior = object(TScroller)
    Constructor Init(R: TRect; SX,SY: PSCrollBar);
    Procedure Draw; virtual;
  end;
{-----}
PROCEDURE TMyApp.Run;
{Читает строки из текстового файла и обеспечивает их просмотр}
var
  R: TRect;
  W: PWindow;
  s,name: string;

```

Рис.23.4. Окно со скроллером

```

Procedure TMyApp.Run;
{Читает строки из текстового файла и обеспечивает их просмотр}
var
  R: TRect;
  W: PWindow;
  s,name: String;
  f: text;
begin
  {Получаем в NAME имя файла с текстом программы;}
  name := copy(ParamStr(0),1,pos('.',ParamStr(0))+'PAS';
  {Создаем коллекцию текстовых строк;}
  Lines := New(PCollection, Init(10,5));
  assign(f,name);
  {$I-}
  reset(f);
  {$I+}
  if IOResult = 0 then
    begin                                {Файл успешно открыт}
      with Lines^ do while not EOF(f) do
        begin
          ReadLn(f,s);
          Insert(NewStr(s))
        end;
      Close(f)
    end
  else                                {Файл не был открыт}
    Lines^.Insert(NewStr('Нет доступа к файлу '+name));

```

```

{Создаем окно со скроллером:}
DeskTop^.GetExtent(R);
W := New(PWindow, Init(R, 'Просмотр файла '+name, 0));
with W^ do
  begin
    GetClipRect(R);
    R.Grow(-1, -1);
    Insert(New(PInterior, Init(R, StandardScrollBar(
      sbHorizontal+ sbHandleKeyboard),
      StandardScrollBar(sbVertical+sbHandleKeyboard))))
  end;
DeskTop^.Insert(W);
{Ждем действий пользователя:}
Inherited Run
end {TMyApp.Run};
{-----}
Constructor TInterior.Init;
{Создает окно скроллера}
begin
  Inherited Init(R, SX, SY);
  GrowMode := gfGrowHiX+gfGrowHiY;
  SetLimit(128, Lines^.Count-1)
end {TInterior.Init};
{-----}
Procedure TInterior.Draw;
{Выводит на экран содержимое окна скроллера}
var
  Y: Integer;
  B: TDrawBuffer;
  S: String;
begin
  for Y := 0 to pred(Size.Y) do
    begin
      MoveChar(B, ' ', GetColor(1), Size.X);
      if (Y+Delta.Y < Lines^.Count) and
        (Lines^.At(Y+Delta.Y) <> NIL) then
        begin
          S := PString(Lines^.At(Y+Delta.Y))^;
          MoveStr(B, copy(S, Delta.X+1, Length(S)-
            Delta.X), GetColor(1))
        end;
      WriteLine(0, Y, Size.X, 1, B)
    end
  end {TInterior.Draw};
{-----}
var
  P: TMyApp;

```

```
begin
  P.Init;
  P.Run;
  P.Done
end.
```

В программе перекрывается метод *TApplication.Run*. В потомке *TMyApp* этот метод вначале считывает текстовые строки из файла с текстом программы в коллекцию *Lines* и создает на экране окно со скроллером. После этого вызывается стандартный метод *TApplication.Run*.

Метод *TInterior.Draw* обеспечивает вывод нужных строк в окно скроллера. Для определения порядкового номера выводимых строк и их положения относительно границ скроллера используется поле *TScroller.Delta*. Обратите внимание: если в коллекцию помещается «пустая» строка, т.е. строка нулевой длины, глобальная функция *NewStr* возвращает значение *NIL*. В методе *TInterior.Draw* оператор

```
if (Y+Delta.Y < Lines^.Count) and
(Lines^.At(Y+Delta.Y) <> NIL) then ...
```

осуществляет проверку значения получаемого из коллекции указателя на *NIL*; если бы мы не предусмотрели эту проверку, прогон программы (использование *NIL*—указателя) на некоторых ПК мог бы привести к аварийному останову.

23.6.6. Просмотр списка файлов

Ниже приводится программа, в которой показано, как можно создать и использовать диалоговое окно для выбора файлов из любого каталога. В пример включены лишь минимальные средства, с помощью которых на экране формируется окно выбора файлов и окно с сообщением о конкретном выборе (см. рис.23.5).

Для реализации просмотра списка файлов и выбора из этого списка нужного файла в Turbo Vision предусмотрен объект *TListBox*. Этот объект создает специальное окно скроллера, содержащее одну вертикальную полосу и указатель на текущий элемент. Имена файлов помещаются в коллекцию строк, указатель на которую передается объекту с помощью метода *TListBox.NewList*.

В программе используются две глобальные переменные, содержащие указатель на коллекцию *L* и номер выбранного элемента *Foc*. В объекте *TApplication* перекрываются методы *Run* и *Done*. Новый метод *TMyApp.Run* создает коллекцию и окно просмотра. Метод *TMyApp.Done* перед завершением работы программы формирует окно, в котором сообщается имя выбранного из списка файла. Заметим, что это имя помещается в переменную *Foc* в момент выхода из программы с помощью перекрываемого метода *TListBox.Valid*.

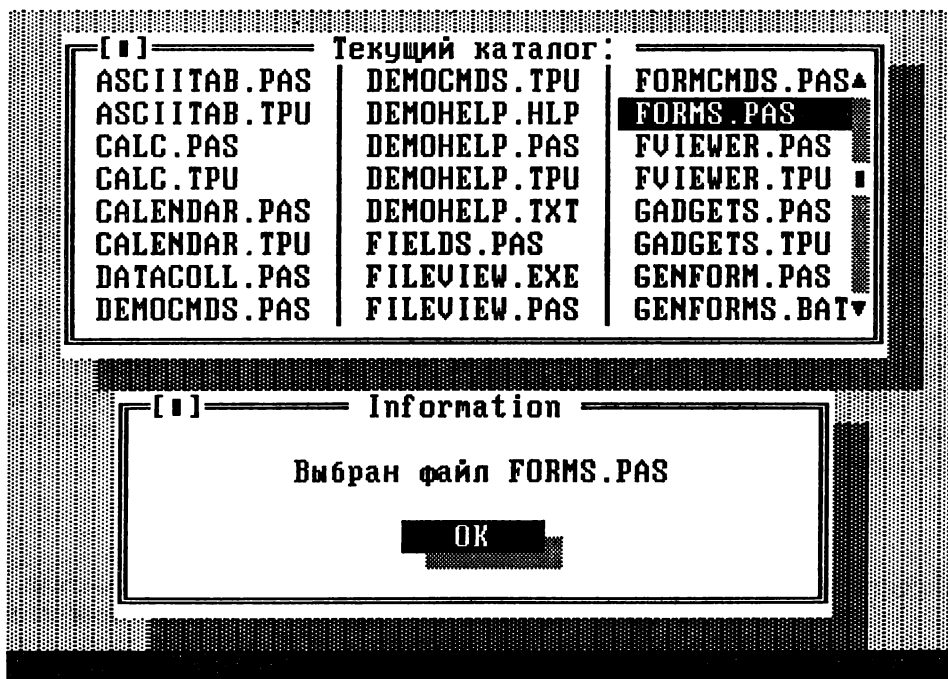


Рис.23.5. Окно выбора файлов

```
{ $X+ }
```

```
Uses DOS, Objects, App, Views, Dialogs, Drivers, MsgBox;
```

```
var
```

```
  L: PStringCollection;           {Коллекция имен файлов}
```

```
  Foc: String;                   {Выбранный файл}
```

```
type
```

```
  TMyApp = object (TApplication)
```

```
    Procedure Run; Virtual;
```

```
    Destructor Done; Virtual;
```

```
  end;
```

```
  PMyListBox = ^TMyListBox;
```

```
  TMyListBox = object (TListBox)
```

```
    Function Valid(Command: Word): Boolean; Virtual;
```

```
  end;
```

```
{-----}
```

```
Procedure TMyApp.Run;
```

```
{Создает диалоговое окно с TListBox}
```

```
var
```

```
  R, RR: TRect;
```

```
  W: PDialog;
```

```
  S: SearchRec;
```

```
  B: PScrollBar;
```

```
  P: PListBox;
```

```

begin
  {Создаем коллекцию имен файлов:}
  L := New(PStringCollection, Init(50,10));
  FindFirst('\games\fl9\*.*', Archive, S);
  While DosError = 0 do with S, L^ do
    begin
      Insert(NewStr(Name));
      FindNext(S)
    end;
  {Создаем окно:}
  R.Assign(17, 4, 63, 14);
  W := New(PDialog, Init(R, 'Текущий каталог:'));
  {Вставляем в окно TListBox:}
  with W^ do
    begin
      RR.Assign(44, 1, 45, 9);
      B := New(PScrollBar, Init(RR));
      Insert(B);
      R.Assign(1, 1, 44, 9);
      P := New(PMyListBox, Init(R, 3, B));
      P^.NewList(L);
      Insert(P)
    end;
    DeskTop^.Insert(W);           {Помещаем окно на экран}
    Inherited Run                 {Ждем команду Alt-X}
  end; {TMyApp.Run}
  {-----}
  Function TMyListBox.Valid;
  {Помещает в Foc имя выбранного файла}
  begin
    Foc := PString(L^.At(Focused))^;
    Valid := True
  end; {TMyListBox.Valid}
  {-----}
  Destructor TMyApp.Done;
  {Выводит имя выбранного файла}
  var
    R: TRect;
  begin
    R.Assign(20, 15, 60, 22);
    MessageBoxRect(R, #3'Выбран файл '+Foc, NIL, $402);
    Inherited Done
  end {TMyApp.Done};
  {-----}
  var
    P: TMyApp;
  begin
    P.Init;

```

```
P.Run;  
P.Done  
end.
```

Окно *TListBox* управляется мышью и клавишами. В частности, клавишами смещения курсора можно выбрать нужный файл, клавишами *PgUp*, *PgDn* листать окно со списком. Работают также клавиши *End*, *Home*, *Ctrl-PgUp*, *Ctrl-PgDn*.

В момент обращения к методу *TMyApp.Done* вызывается функция *TMyListBox.Valid*, которая определяет номер выделенного файла (этот номер хранится в поле *TListBox.Focused*) и переписывает имя этого файла из коллекции в глобальную переменную *Foc*.

ПРИЛОЖЕНИЯ

П1. СРЕДА ТУРБО ПАСКАЛЯ

П1.1. ЭЛЕМЕНТЫ ДИАЛОВОЙ СРЕДЫ

При работе с Турбо Паскалем на экране ПК может формироваться сложная структура прямоугольных участков экрана, рассматриваемых как единое целое и предназначенных для тех или иных действий. Такие участки в тексте книги называются *меню*, *окнами*, *полями* и т.д. На рис.П1.1 показаны эти элементы и даны их названия.

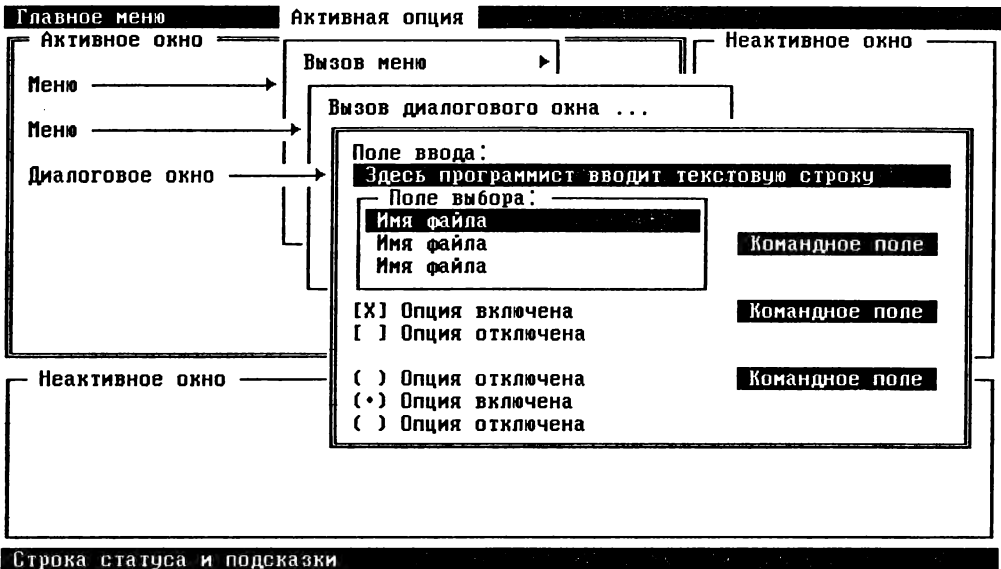


Рис.П1.1. Элементы диалоговой среды Турбо Паскаля

Меню будем называть прямоугольный участок экрана, содержащий кодовые слова и предназначенный для диалогового выбора продолжения работы. Меню фиксирует некоторое текущее состояние диалоговой среды и предлагает несколько альтернативных путей перехода из этого состояния. Содержащиеся в меню кодовые слова обозначают возможные альтернативы. Условимся называть их в дальнейшем *опциями* (option — выбор).

Окно в Турбо Паскале предназначено для обмена информацией между программистом и средой. В окно редактора программист помещает текст программы, в окне программы среда показывает результат ее прогона, в справочном окне появляются справочные сообщения, в отладочном окне программист может наблюдать за изменением переменных в процессе отладки программы. Одновременно на экране может присутствовать сразу несколько окон, однако только одно из них активно в каждый момент. Активное окно очерчивается двойной рамкой, неактивные — одинарной.

Диалоговое окно разворачивается на экране по мере надобности. С помощью диалогового окна пользователь уточняет выбранное действие. Внутри диалогового окна может быть несколько полей. В поле ввода программист может подготовить текстовую строку (например, имя файла). В поле выбора среда предлагает несколько возможных имен файлов. В поле переключаемой опции можно установить необходимую настройку среды, с помощью командных полей — передать среде ту или иную команду.

Работа с меню и окнами значительно упрощается, если Ваш ПК оснащен устройством ввода типа «мышь». Условимся в дальнейшем вместо длинного "устройство ввода типа

«мышь»" писать просто мышь, что соответствует обиходному названию этого прибора. Подвигав мышь по столу, Вы тут же заметите перемещающийся по экрану указатель мыши. С помощью этого указателя можно выбрать нужный элемент диалоговой среды. Для этого подведите к нему указатель и нажмите левую кнопку мыши.

П1.1.1. Работа с окнами

В среде Турбо Паскаля используется несколько окон, которые могут частично или полностью накладываться друг на друга. Для последовательной смены окон используется клавиша F6: нажатие на эту клавишу делает активным очередное окно; если это окно было закрыто другими окнами, оно накладывается поверх них. Для смены активности окон можно использовать и мышь: переместите указатель мыши внутрь неактивного окна и нажмите левую кнопку.

В верхней части рамки, очерчивающей активное окно, имеются два небольших поля, используемых при работе с мышью. Поле  служит для удаления окна с экрана, поле  —

для распахивания окна на весь экран, а если это поле помечено значком , — для возврата к прежним размерам после распахивания. Эти же действия выполняются и с помощью клавиатуры: F5 распахивает окно или возвращает окну обычный размер, Alt-F3 закрывает окно (удаляет его с экрана). Закрытое окно удаляется из системы окон Турбо Паскаля и его уже нельзя вызвать с помощью F6. Перед закрытием окна редактора, содержащего несохраненный на диске текст, среда спросит, нужно ли его сохранить.

Положение и размеры активного окна можно изменять по своему вкусу. При работе с клавиатурой для изменения этих параметров используется команда Ctrl-F5. После этой команды изменяются цвет и линии рамки: таким образом среда сигнализирует о переходе к режиму настройки. Теперь клавишами смещения курсора Вы можете перемещать окно по экрану и этими же клавишами, нажатыми в сочетании с клавишей Shift, можно менять размеры окна. После того, как положение и размеры окна установлены нужным образом, нажимается клавиша Enter. При работе с мышью для изменения положения активного окна следует подвести указатель к левому верхнему углу рамки окна и нажать левую кнопку — окно будет «схвачено» мышью, о чем свидетельствует изменение цвета и линий рамки. Затем, не отпуская кнопки, перемещайте мышь по столу, при этом окно будет смещаться по экрану. Передвинув окно нужным образом, отпустите кнопку. Для изменения размеров окна используется правый нижний угол рамки, который «захватывают» мышью точно так же, как и при перемещении окна.

В окнах редактора и справочной службы содержится текст, который может не помещаться целиком в окне. Правая и нижняя рамки такого окна содержат указатели размеров, в которых показывается положение демонстрируемого в окне фрагмента относительно полных размеров текста. Эти поля можно использовать для того, чтобы перемещать окно относительно текста с помощью мыши. Чтобы сдвинуть окно вниз на одну строку, подведите указатель мыши к самому нижнему краю вертикального указателя размеров так, чтобы он попал на поле , и нажмите левую кнопку. Для сдвига окна вниз на страницу установите указатель мыши в любое место вертикального указателя размеров несколько ниже его середины и нажмите левую кнопку. Точно так же поступают при перемещении окна вверх, вправо и влево, используя для этого соответственно верхнюю часть вертикального указателя и правую и левую части горизонтального указателя, если окно смещается на страницу, или поля , ,  — при смещении на одну строку или символ.

П1.1.2. Работа с меню

Для перехода из состояния редактирования к выбору из главного меню используется клавиша F10, для возврата в редактор — клавиша Esc. В активном меню указателем (цветом или оттенком) выделяется очередная опция. Для выбора того или иного продолжения переместите клавишами смещения курсора указатель к нужной опции и нажмите Enter. Выбирать можно и другим способом. В кодовом слове опции цветом выделяется одна из букв. Для выбора опции нажмите клавишу с нужной буквой, если выбирается опция из дополнительного меню, или комбинацию Alt-*<буква>*, если выбирается опция главного меню. При работе с мышью для выбора из меню нужно переместить указатель мыши к соответствующей опции и нажать левую кнопку.

Выбор опции обычно приводит к разворачиванию нового меню или диалогового окна. Если справа от опции стоит многоточие, эта опция связана с разворачиванием диалогового окна, если стоит значок , вызывается дополнительное меню.

Некоторые часто используемые опции, содержащиеся в дополнительных меню, можно вызвать непосредственно из режима редактирования. Справа от таких опций в меню указывается клавиша или комбинация клавиш, которая позволяет сделать это.

Следует учесть, что детальную информацию на английском языке о том или ином предложении (опции) можно получить с помощью справочной службы, если клавишами перемещения курсора сместить указатель к этой опции и нажать клавишу *F1*.

П1.1.3. Работа с диалоговым окном

С помощью диалогового окна уточняется выбранное действие. В диалоговом окне имеется несколько полей, в которых группируется информация о возможностях среды и ее настройке.

Сразу после разворачивания диалогового окна активизируется то или иное поле, которое выделяется цветом (оттенком). В активных полях ввода, переключаемых опций или выбора файла, кроме того, виден мигающий курсор.

Запомним следующие правила:

- для перехода от одного поля к другому предназначена клавиша табуляции *Tab* (клавиша располагается в левой верхней части основной зоны клавиатуры и обозначена двумя разнонаправленными стрелками);
- для перехода внутри поля используются клавиши смещения курсора;
- закрыть диалоговое окно можно клавишей *Esc* (в этом случае не происходит никаких действий, связанных с окном) или клавишей *Enter* (в этом случае выполняются все указанные в окне установки или выбирается указанный файл).

При работе с мышью для выбора поля или опции следует сместить к этому полю (опции) указатель мыши и нажать левую кнопку. Для того, чтобы закрыть окно и выполнить все связанные с ним установки, нужно указать мышью на соответствующее командное поле и нажать левую кнопку (в большинстве случаев это поле помечается символами *OK*). Чтобы закрыть диалоговое окно и не выполнять никаких действий, используется поле  верхней рамки или командное поле со словом *Cancel* (отмена).

Если по смыслу того или иного исполняемого действия необходимо ввести текстовую строку (например, имя файла), то сразу после раскрытия диалогового окна активизируется поле ввода с мигающим курсором. Следует ввести нужный текст и нажать *Enter*. При вводе текста используются правила редактирования, принятые в редакторе Турбо Паскаля: ошибочно введенный символ можно стереть клавишами *Backspace* или *Del*, причем этот символ может быть в любом месте строки (используйте клавишу перевода курсора влево, чтобы указать ошибочный символ, или укажите на него мышью и нажмите левую кнопку); ввод текста может происходить в режиме замены (переключается клавишей *Ins*). Если после подготовки текста Вы нажмете *Enter*, текст будет введен и диалоговое окно закроется, однако если Вы по каким-либо причинам измените свое решение, достаточно нажать *Esc*, чтобы закрыть диалоговое окно без ввода текста. Можно также повторить ранее введенный в аналогичной ситуации текст, например, уже вводившееся имя файла. Для этого вместо ввода

ожидаемого текста нажмете клавишу смещения курсора вниз или укажите на поле  справа от поля ввода мышью и нажмите ее левую кнопку. В развернувшемся на экране небольшом окне содержится протокол использования данной опции. В протоколе сохраняются текстовые строки, введенные ранее, и можно клавишами смещения курсора подвести цветной указатель — прямоугольник к нужной Вам строке протокола и нажать *Enter* или указать на эту строку мышью и дважды подряд нажать на левую кнопку (нажимать на кнопку нужно быстро, без заметной паузы, иначе среда не воспримет это как команду ввода и просто переместит указатель к нужной строке протокола).

Переключаемые опции задают выбор нужной настройки среды из двух или нескольких вариантов. Варианты могут быть связаны с включением или отключением какого-либо параметра среды. Например, можно потребовать от компилятора использовать арифметический сопроцессор или не использовать его. Слева от таких опций в диалоговом окне имеется небольшое поле выбора, выделенное квадратными скобками; включенный параметр отмечается символом *X* в этом поле: [*X*]; если поле пустое [], данный параметр не задействован. Если переключаемая опция задает выбор из нескольких вариантов, слева от указателя каждого

варианта имеется поле выбора, выделенное двумя круглыми скобками, причем выбранный вариант отмечается точкой: (*).

Чтобы изменить состояние переключаемого параметра или выбрать другой его вариант, нужно клавишами смещения курсора подвести указатель к соответствующей опции и нажать клавишу *Пробел* (длинная клавиша в самом низу клавиатуры) или подвести указатель мыши к полю выбора слева от опции и нажать левую кнопку. Можно также нажать клавишу с буквой, которая выделена цветом в нужной опции.

В диалоговом окне обязательно имеется несколько командных полей, которые располагаются в правой или нижней части окна и выделяются цветом. С каждым таким полем связана некоторая команда. Эту команду можно выполнить, если активизировать поле клавишей табуляции и нажать *Enter* или указать на него мышью и нажать левую кнопку.

П1.2. СИСТЕМА МЕНЮ

Все управление средой Турбо Паскаля осуществляется в основном с помощью системы последовательно разворачивающихся меню. Лишь одно из них — главное меню — постоянно присутствует на экране, остальные разворачиваются по мере выбора продолжений.

Главное меню содержит фактически лишь оглавление дополнительных меню. В этих меню сгруппированы близкие по своему роду действия, условное название которых и служит кодовым словом соответствующей опции главного меню:

File (файл) — действия с файлами и выход из системы;

Edit (редактировать) — восстановление испорченной строки и операции с временным буфером;

Search (искать) — поиск текста, процедуры, функций или места ошибки;

Run (работа) — прогон программы;

Compile (компилировать) — компиляция программы;

Debug (отладка) — отладка программы;

Tools (инструменты) — вызов вспомогательных программ (утилит);

Options (варианты) — установка параметров среды;

Window (окно) — работа с окнами;

Help (помощь) — обращение к справочной службе.

Ниже описываются опции дополнительных меню Турбо Паскаля.

П1.2.1. Меню опции FILE

NEW. Создает и открывает новое окно редактора с именем *NONAMExx.PAS*. Порядковый номер *XX* окна зависит от количества окон со стандартным именем *NONAME*, открытых к моменту обращения к опции.

OPEN. Открывает новое окно редактора и помещает в него указанный дисковый файл. При обращении к этой опции открывается диалоговое окно (рис.П1.2), в поле ввода которого можно написать нужное имя файла. Если в имени опущено расширение, среда добавит стандартное расширение *.PAS*. Имени файла может предшествовать путь. Нужный файл Вы можете также выбрать из поля выбора, предварительно активизировав это поле мышью или клавишей *Tab*; при работе с мышью для загрузки нужного файла из списка в поле выбора укажите на имя файла мышью и дважды подряд с небольшим интервалом нажмите левую кнопку. Вы можете открыть доступ к протоколу использования этой опции и выбрать в этом протоколе одно из ранее использованных имен. Для этого при активном поле ввода нажмите



клавишу смещения курсора вниз или укажите мышью на поле  справа от поля ввода и нажмите ее левую кнопку. В открывшемся окне протокола подведите указатель клавишами смещения курсора к нужной строчке и нажмите *Enter*. Командное поле *OPEN (открыть)* используется для команды чтения файла в новое редакторское окно, *REPLACE (заменить)* — для замены существующего в активном редакторском окне текста на текст, считанный из файла.

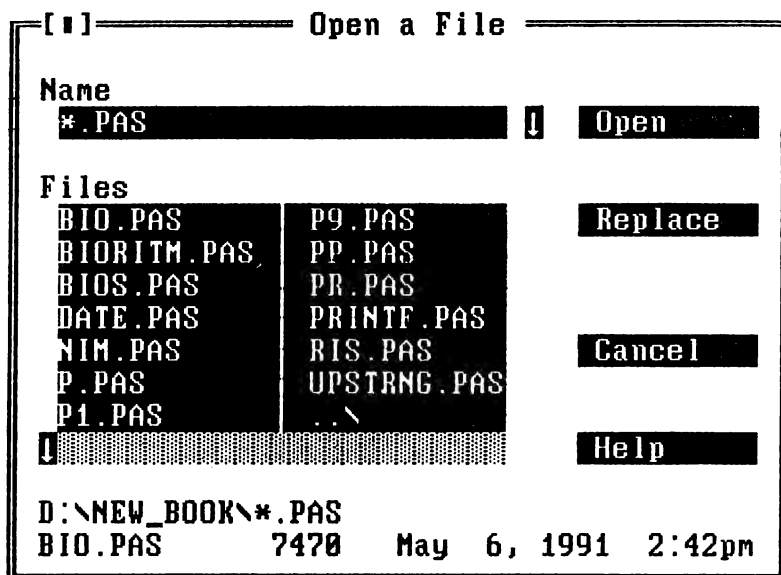


Рис.П1.2. Диалоговое окно опции File/Open

Опция вызывается непосредственно из редактора клавишей F3.

SAVE. Записывает содержимое активного окна редактора в дисковый файл. Если это окно связано с именем *NONAMExx.PAS*, среда запросит новое имя файла (см. ниже опцию *SAVE AS*). Опция вызывается непосредственно из редактора клавишей F2.

SAVE AS. Записывает содержимое активного окна редактора в дисковый файл под другим именем. Диалоговое окно этой опции изображено на рис.П1.3. В поле ввода Вы должны написать имя того файла, в который будет переписано содержимое активного окна редактора. Вы можете выбрать уже существующий файл из поля выбора или из протокола опции. В этом случае в зависимости от настройки среды старое содержимое файла будет уничтожено или сохранено в виде страховочной копии с расширением *.BAK* (настройку среды см. в опции *OPTIONS/ENVIRONMENT*).

SAVE ALL. Записывает содержимое всех окон редактора в соответствующие дисковые файлы.

CHANGE DIR. Позволяет изменить текущий каталог пользователя. В поле выбора диалогового окна этой опции (рис.П1.4) приводится дерево каталогов текущего диска.

Перемещаясь по этому дереву, можно указать на нужный каталог, после чего с помощью командного поля *CHDIR* (*C*Hange *D*irectory — изменить каталог) сменить текущий каталог. Если выбран указатель *DRIVES* (*д*исководы), можно изменить также текущий диск. Командное поле *REVERT* (*в*озвращаться) позволит восстановить прежний текущий каталог, если Вы по каким-либо причинам решите отказаться от сделанного Вами изменения (эта команда действует до момента закрытия окна).

PRINT. Печатает содержимое активного окна редактора на принтере или выводит его в файл (см. опцию *PRINTER SETUP*).

PRINTER SETUP. Настраивает среду на печать текущего файла. Поле ввода *Filter path* должно содержать имя программы-фильтра *PRNFLT.RXE* и, возможно, путь к этой программе. Поле ввода *Command line* содержит выбор принтера/файла и параметров печати. Турбо Паскаль поддерживает три типа принтеров: матричные принтеры *Epson*, лазерные принтеры *HP LaserJet* и так называемые постскрипт-принтеры, т.е. принтеры, «понимающие» язык описания страниц *PostScript*. Для указания нужного типа принтера в поле *Command line* помещаются следующие строки:

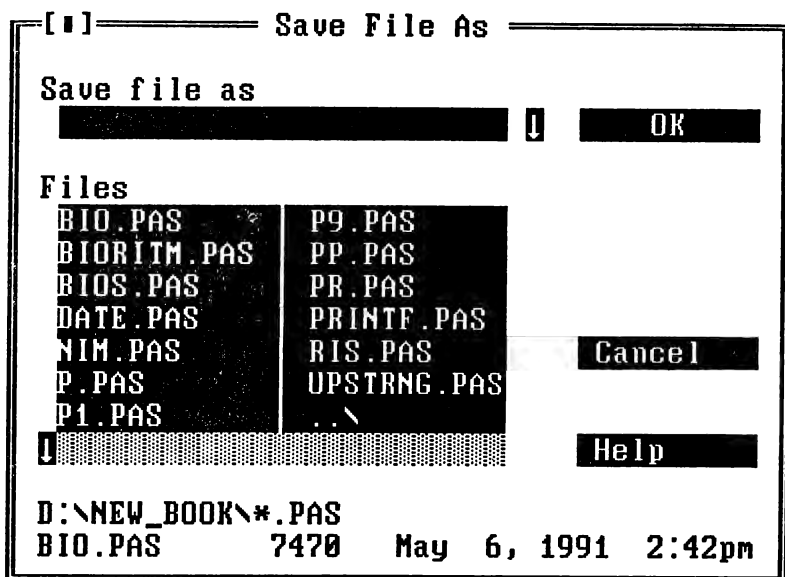


Рис.П1.3. Диалоговое окно опции File/Save as

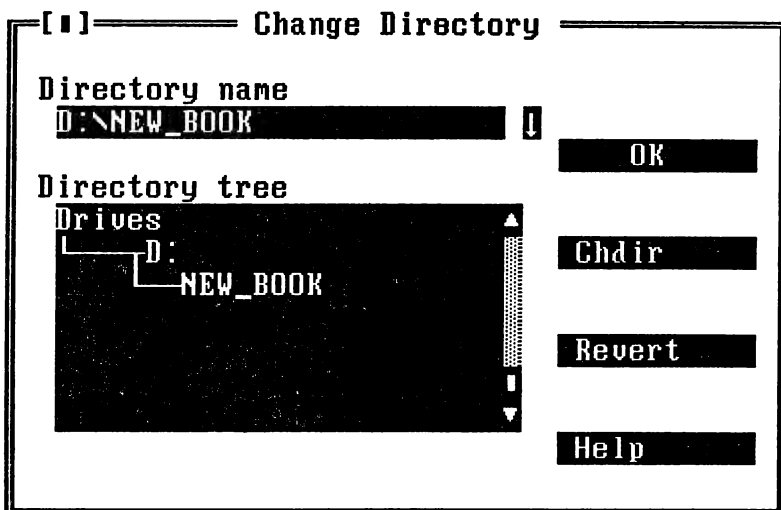


Рис.П1.4. Диалоговое окно опции File/Change dir

Строка	Смысл
\$NOSWAP /ASCII	Неизвестный тип принтера; в выводной поток помещаются только текстовые символы и коды перевода строки
\$NOSWAP /EPSON	Принтер Epson или совместимый с ним; система команд Epson поддерживается большинством матричных принтеров других фирм
\$NOSWAP /HP	Принтер HP LaserJet или совместимый с ним; система команд HP LaserJet поддерживается большинством лазерных принтеров других фирм
\$NOSWAP /PS	Постскрипт-принтер, воспринимающий команды на аппаратно-независимом языке описания страниц PostScript

\$NOSWAP /Охххх	Вывод в файл; имя файла XXXX должно следовать сразу за символами /О без пробелов; если файл не существует, он будет создан в процессе вывода
Эти команды могут дополняться следующими параметрами:	
/Рхххх	Определяет количество строк на одной странице (по умолчанию 55)
/Тхххх	Количество символов пробела, заменяющих один символ табуляции (по умолчанию 8)

DOS SHELL. Обеспечивает временный выход в ДОС. Турбо Паскаль остается резидентным в оперативной памяти и занимает значительную ее часть, поэтому под управлением ДОС в этом состоянии могут выполняться только сравнительно небольшие по объему программы. Чтобы увеличить объем свободной памяти для ДОС, необходимо перед обращением к этой опции сбросить режим отладки клавишами **Ctrl-F2**. Для возврата в Турбо Паскаль нужно в ответ на запрос ДОС напечатать слово **EXIT** и нажать **Enter**.

EXIT. Завершает работу с Турбо Паскалем. Опция вызывается непосредственно из редактора командой **Alt-X**.

П1.2.3. Меню опции EDIT

UNDO. В активном окне редактора восстанавливает только что уничтоженную командой **Ctrl-Y** или измененную строку. Турбо Паскаль создает специальный буфер изменений для каждой страницы редактора. Последовательное использование опции **UNDO** может отменить все сделанные Вами изменения текста. Опция может вызываться непосредственно из окна редактора клавишами **Alt-Backspace**.

REDO. Отменяет действие предыдущей команды **UNDO**.

CUT. Удаляет выделенный блок из окна редактора и переносит его в буфер обмена Clipboard (команды редактора для работы с блоками см. в гл.1). Опция вызывается непосредственно из редактора командой **Shift-Del**.

COPY. Копирует выделенный блок из окна редактора в буфер обмена Clipboard. Опция вызывается непосредственно из редактора командой **Ctrl-Ins**.

PASTE. Копирует содержимое буфера обмена Clipboard в окно редактора. Содержимое буфера остается без изменений и может использоваться повторно. Опция вызывается непосредственно из редактора командой **Shift-Ins**.

CLEAR. Удаляет из окна редактора выделенный блок, но не помещает его в буфер. Удаленный фрагмент безвозвратно теряется. Опция вызывается непосредственно из редактора командой **Ctrl-Del**.

SHOW CLIPBOARD. Показывает содержимое буфера обмена.

П1.2.4. Меню опции SEARCH

FIND. Обеспечивает поиск нужного фрагмента текста в активном окне редактора. В момент обращения к этой опции в поле выбора диалогового окна (рис.П1.5) содержится слово, на которое указывал курсор в активном окне редактора. Вы можете ввести новое слово или текстовую строку, положение которой в редактируемом файле Вам необходимо найти, или выбрать эту строку из протокола. После нажатия на **Enter** (или выбора командного поля ОК) редактор отыщет этот фрагмент в тексте и установит курсор на его начало.

Поиск управляется следующими переключаемыми опциями:

Case sensitive — учитывать величину букв (т.е. прописные буквы считать отличающимися от строчных; опция определена только для латинских букв);

Whole words only — искать по совпадению целых слов (если текст будет обнаружен внутри более длинного слова, поиск продолжится дальше);

Regular expression — искать по выражению — описателю текста (см. ниже);

Forward — направление поиска вниз по тексту;

Backward — направление поиска вверх по тексту;

Global — искать во всем тексте;

Selected text — искать только в выделенном блоке;

From cursor — начать поиск от текущего положения курсора;

Entire scope — искать от начала текста.

[]
Find

Text to find

1

Options

☐ Case sensitive

☐ Whole words only

☐ Regular expression

Direction

☒ Forward

☐ Backward

Scope

☒ Global

☐ Selected text

Origin

☐ From cursor

☒ Entire scope

OK

Cancel

Help

Рис.П1.5. Диалоговое окно опции Search/Find

В поле ввода можно ввести выражение — описатель текста, которое формируется из следующих специальных символов:

^ — в начале строки в выражении — описателе означает начало текстовой строки в ис — комом тексте;

\$ — в конце строки в выражении — описателе показывает конец текстовой строки;

— на этом месте может стоять любой символ;

* — после любого символа означает любое количество (в том числе ноль) этих символов, которые могут стоять вместо него; например, b* означает b, bo, boo, bot, be;

+ — после символа означает один или больше (не ноль) этих символов, которые могут стоять вместо него; например, b+ означает bo, bot, boo, но не b или be;

[] — означает один из символов, который может стоять внутри скобок, но не любой другой символ; например, [bot] означает b, o или t;

[^] — символ ^ в начале строки, заключенной в квадратные скобки, означает отрицание; например [^bot] — это любые символы, кроме b, o или t;

[-] — символ «-» между двумя символами, обрамленными квадратными скобками, оп — ределяет диапазон; например, [b-o] означает любые символы от b до o включительно;

\ — перед специальным символом означает сам символ; например, \^ означает сам символ ^, а не начало строки.

REPLACE. Отыскивает в окне редактора нужный текстовый фрагмент и заменяет его на новый. Диалоговое окно этой опции похоже на предыдущее со следующими отличиями:

- в поле ввода *Text to find* нужно ввести искомый текст или выражение — описатель, в поле *New text* — тот текст, который будет вставляться вместо искомого;
- переключаемая опция *Prompt on replace* указывает на необходимость запросить под — тверждение у программиста, прежде чем производить замену найденного фрагмента;
- командное поле *Change all* используется для поиска и замены всех обнаруженных вхождений фрагмента текста (поле *OK* или нажатие на *Enter* означает только одно — кратный поиск — замену).

SEARCH AGAIN. Повторяет поиск или поиск и замену фрагмента текста для ранее уста — новленных параметров.

GO TO LINE NUMBER. Осуществляет позиционирование курсора в окне редактора на строку с указанным номером.

SHOW LAST COMPILER ERROR. Показывает строку текста программы, в которой была обнаружена синтаксическая ошибка при последнем прогоне компилятора.

FIND ERROR. Отыскивает в тексте программы строку, вызвавшую ошибку периода исполнения программы. Как правило, при работе в среде Турбо Паскаля ошибка периода прогона программы (например, деление на ноль) вызывает автоматическое прекращение прогона, в окне редактора появляется текст программы и курсор показывает то место, в котором возникла эта ошибка. Если программа компилировалась без привлечения средств отладки (см. ниже опцию *OPTIONS/COMPILER*) или исполнялась вне среды Турбо Паскаля, нужно записать или запомнить два шестнадцатиричных числа, которые появляются на экране в сообщении об ошибке и указывают адрес ошибочной ситуации (задаются в формате *SSSS:OOOO*, где *SSSS* — сегмент, а *OOOO* — смещение), затем загрузить Турбо Паскаль (если программа исполнялась вне среды) и вызвать эту опцию. После ввода адреса ошибки, среда начнет поиск ошибочного оператора.

FIND PROCEDURE. Позволяет в режиме отладки отыскать в тексте программы нужную процедуру или функцию.

П1.2.5. Меню опции RUN

RUN. Осуществляет компиляцию, компоновку и исполнение (прогон) программы из файла редактора. Компиляция проходит в режиме *MAKE* (см. ниже опцию *COMPILE/MAKE*). Если программа уже откомпилирована к этому моменту, то среда сразу начнет ее прогон. Опция вызывается непосредственно из редактора командой *Ctrl-F9*.

GO TO CURSOR. Начинает или продолжает режим отладки исполняемой программы под управлением встроенного отладчика. Вначале осуществляются все действия по компиляции и компоновке программы, затем программа начинает работать обычным образом (экран переходит в режим воспроизведения окна программы) и останавливается перед выполнением первого оператора из той строки, на которую указывает курсор. В этот момент экран возвращается в режим воспроизведения окна редактора, а строка с курсором выделяется цветным прямоугольником. Можно перевести курсор к новой строке и вновь выбрать эту опцию — программа остановится перед выполнением нового оператора и т.д. В этом режиме доступны все средства встроенного отладчика. Для прекращения отладки нажмите клавиши *Ctrl-F2*. Опция вызывается непосредственно из редактора клавишей *F4*.

TRACE INTO. Начинает или продолжает режим отладки исполняемой программы под управлением встроенного отладчика. Если к моменту обращения к этой опции режим отладки не был запущен, он запускается точно так, как если бы была вызвана опция *GO TO CURSOR*, однако программа останавливается перед первым исполняемым оператором, т.е. указатель будет указывать на слово *BEGIN*, открывающее раздел операторов основной программы. Если режим отладки уже был запущен, вызов этой опции приведет к выполнению всех действий, запрограммированных в текущей строке, и указатель сместится к следующей строке программы. Если текущая строка содержит обращение к процедуре или функции, управление будет передано внутрь этой процедуры (функции) и программа остановится перед исполнением ее первого оператора. Таким образом, с помощью этой опции можно по шагам проследить исполнение всех нестандартных процедур (функций). Опция вызывается непосредственно из редактора клавишей *F7*.

STEP OVER. Также, как и предыдущая опция, начинает или продолжает пошаговое проследивание работы программы, но не проследивается работа вызываемых процедур и функций. Опция вызывается непосредственно из редактора клавишей *F8*.

PROGRAM RESET. Сбрасывает все ранее задействованные отладочные средства и прекращает отладку программы. Удаляет исполнявшуюся программу из памяти и закрывает все открытые в ней в этот момент файлы. Опция вызывается непосредственно из редактора командой *Ctrl-F2*.

PARAMETERS. Позволяет задать текстовую строку параметров, которые ДОС передает вызываемой программе. Эта строка передается программе, находящейся в окне редактора, при ее прогоне.

П1.2.6. Меню опции COMPILE

COMPILE. Компилирует программу или модуль, который загружен в данный момент в активное окно редактора. Если в этой программе (модуле) содержатся обращения к нестандартным модулям пользователя, последние уже должны быть откомпилированы и храниться на диске в виде *TPU*-файлов. Опция вызывается непосредственно из редактора командой *Alt-F9*.

MAKE. Создает программу, которая, возможно, содержит включаемые файлы и/или обращения к нестандартным модулям. Прежде всего компилируется начальный файл, если, разумеется, он определен опцией *COMPILE /PRIMARY FILE* (см. ниже). Если начальный файл не задан, компилируется файл из активного окна редактора. Если в процессе компиляции встретилось объявление нестандартного модуля, среда проверяет, были ли сделаны в соответствующем *PAS*-файле с текстом программы этого модуля какие-либо изменения с момента последней его компиляции и получения *TPU*-файла; если изменения были, *TPU*-файл создается вновь, а если изменения коснулись его интерфейсной части, будут перекомпилированы также все другие объявленные в программе модули, в которых содержатся обращения к измененному модулю. Однако, если *PAS*-файл с текстом измененного модуля не будет найден, система воспользуется существующим *TPU*-файлом без контроля его «свежести». Отметим, что этот контроль осуществляется по дате и времени создания *PAS*-файла и соответствующего ему *TPU*-файла. Если системная дата установлена неправильно, среда Турбо Паскаля может ошибочно откомпилировать *PAS*-файл, для которого существует адекватный ему *TPU*-файл.

Опция существенно упрощает процесс разработки многофайловых программ, так как всегда компилируется только тот минимум файлов, которых коснулись сделанные в программе изменения. Опция вызывается непосредственно из редактора клавишей *F9*.

BUILD. Эта опция полностью подобна опции *MAKE* за одним исключением: для всех *TPU*-файлов отыскивается соответствующий *PAS*-файл и осуществляется его перекомпиляция независимо от того, были ли сделаны в нем изменения или нет. После компиляции в этом режиме, Вы можете быть уверены в том, что в полученной программе учтены все изменения.

DESTINATION. Эта опция управляет выходом компилятора: если справа от нее стоит кодовое слово *Memory* (память), выходной файл компилятора будет сохранен в оперативной памяти и может затем сразу же запускаться из Турбо Паскаля без его загрузки с диска; если справа стоит кодовое слово *Disk* (диск), файл с кодом программы будет сохранен на диске в виде файла с расширением *.EXE*. Если объявлен начальный файл, его имя будет присвоено имени вновь создаваемого *EXE*-файла, в противном случае *EXE*-файл получит имя файла из того окна редактора, которое содержит текст основной программы. Независимо от значения этого параметра *TPU*-файлы, создаваемые в режимах *RUN/RUN*, *COMPILE/MAKE* и *COMPILE/BUILD*, будут помещены на диск.

PRIMARY FILE. Задаёт имя начального файла. Если это имя задано, то вне зависимости от того, какая часть программы загружена в данный момент в окна редактора, ее компиляция в режимах *RUN*, *MAKE* и *BUILD* будет начинаться с этого файла. Чаще всего начальный файл содержит текст основной части программы. В этом случае при загрузке в окно (окна) редактора включаемого файла или файла-модуля компилятор сумеет правильно построить программу. Если начальный файл не указан, то компиляция в режимах *RUN*, *MAKE* и *BUILD* возможна только в том случае, когда в активном окне редактора находится основная программа.

CLEAR PRIMARY FILE. Очищает имя начального файла, заданное опцией *PRIMARY FILE*.

INFORMATION. Показывает статистику программы.

П1.2.7. Меню опции DEBUG

BREAKPOINTS. Эта опция позволяет просмотреть все контрольные точки и при необходимости удалить, переместить любую контрольную точку или задать условия ее работы. В диалоговом окне опции (рис.П1.6) приводится список всех контрольных точек с указанием имени файла (колонок *Breakpoints list*), номера строки в этом файле (колонок *Line #*), с которой связана контрольная точка, условия, при котором срабатывает останов программы в этой точке (колонок *Condition*), и количества проходов (колонок *Pass*), в течение которых останов не происходит. Цветным указателем выделяется текущая контрольная точка.

[!] Breakpoints

Breakpoints list	Line #	Condition	Pass
BIO.PAS	48		0
BIO.PAS	74		0

OK Edit Delete View Clear all Help

Рис.П1.6. Диалоговое окно опции Debug/Breakpoints

В качестве условия, управляющего работой контрольной точки, можно указать любое допустимое условное выражение, которое будет вычисляться в ходе исполнения программы; контрольная точка будет игнорироваться до тех пор, пока значение этого выражения не окажется равным *TRUE*. Вычисление выражения и останов будут происходить только тогда, когда с момента запуска программы будет выполнено заданное количество обращений к строке с контрольной точкой (колонок *Pass*).

С помощью командного поля *Edit* можно отредактировать текущую точку, т.е. установить новые ее параметры (файл, номер строки, условие и количество проходов). Задав новый файл и/или номер строки, можно переместить точку на новое место. Командное поле *Delete* используется для удаления текущей точки, а поле *Clear all* — для удаления всех контрольных точек. С помощью поля *View* можно загрузить в окно редактора и установить его содержимое так, чтобы увидеть соответствующую контрольную точку.

CALL STACK. Делает активным окно программногo стека. В этом окне отображаются все вызовы процедур и функций. Внизу стека находится *PROGRAM*, т.е. имя Вашей программы, в вершине стека — текущая процедура (функция). Каждое новое обращение к процедуре (функции) отображается в этом окне в виде имени подпрограммы и списка параметров вызова. Эта опция вызывается из редактора командой *Ctrl-F3*.

REGISTER. Делает активным окно регистров. В этом окне отображается текущее состояние всех регистров микропроцессора ПК.

WATCH. Делает активным окно отладки.

OUTPUT. Делает активным окно программы.

USER SCREEN. Делает активным окно программы и распаивает его на весь экран. Вызывается из редактора командой *Alt-F5*.

EVALUATE/MODIFY. Эта опция дает возможность в процессе отладки просмотреть содержимое любой переменной или найти значение любого выражения. При необходимости можно с ее помощью установить новое значение любой переменной. При обращении к ней на экране разворачивается диалоговое окно, содержащее три поля: *EXPRESSION* (выражение), *RESULT* (результат) и *NEW VALUE* (новое значение) (рис.П1.7).

[] Evaluate and Modify	
Expression	
	↓ Evaluate
Result	
	Modify
	Cancel
New value	
	↓ Help

Рис.П1.7. Диалоговое окно опции Debug/Evaluate/Modify

В первом поле следует ввести имя любой переменной или некоторое выражение. Сразу после того, как Вы нажмете *Enter*, в поле *RESULT* появится соответствующее значение или сообщение *Unknown identifier* (неопределенный идентификатор), если такая переменная не определена в Вашей программе. К моменту вызова опции программа должна находиться в режиме отладки, в противном случае это сообщение будет даваться для любых переменных и выражений с их участием.

Если Вы запросили значение переменной, Вы можете перевести курсор в нижнее поле *NEW VALUE* и установить новое значение переменной — это значение будет немедленно передано в программу.

При обращении к опции среда анализирует ближайшее окружение курсора в активном окне редактора и, если это возможно, выделяет идентификатор или константу, на которую указывает курсор. Выделенное автоматически переносится в поле *EXPRESSION* и предлагается в виде вычисляемого выражения. Таким образом, если перед вызовом этой опции установить курсор на интересующий Вас идентификатор, останется лишь нажать на *Enter*, чтобы тут же получить его значение в поле *RESULT*. Если предлагаемый идентификатор Вас не устраивает, можно его отредактировать или ввести новый. Для ввода нового нажмите на любую алфа-цифровую клавишу, и предлагаемый в окне идентификатор исчезнет, заменившись вновь введенным символом. Для перехода к редактированию предлагаемого идентификатора следует сразу же после появления окна нажать *Home* или *End*, затем перевести курсор к нужному месту идентификатора и отредактировать его с использованием клавиш *Ins*, *Del* и *Backspace*. Наконец, если сразу после вызова опции нажать *End*, а затем — клавишу перевода курсора вправо, появившийся в поле *EXPRESSION* идентификатор дополнится символом, расположенным справа от него в тексте программы. Теперь при каждом нажатии на клавишу перевода курсора вправо очередной символ из текста программы будет копироваться в поле *EXPRESSION*. Описанная возможность существенно облегчает ввод длинных выражений и составных идентификаторов.

Вы можете ввести (и вычислить) выражения с участием констант и переменных из Вашей программы, а также некоторых стандартных функций. Выражения составляются по правилам формирования выражений Турбо Паскаля, причем тип выражения может быть любым стандартным или определенным в программе типом. Фактически единственным ограничением на выражения является то, что в них нельзя использовать вызовы нестандартных функций (можно использовать только следующие предварительно определенные функции и константы: *ABS*, *ADDR*, *CHR*, *DSEG*, *HI*, *IORESULT*, *LENGTH*, *LO*, *MAXAVAIL*, *MEMAVAIL*, *ODD*, *OFS*, *ORD*, *PRED*, *PTR*, *ROUND*, *SEG*, *SIZEOF*, *SPTR*, *SSEG*, *SUCC*, *SWAP*, *TRUNC*, а также массивы *MEM*, *MEMW* и *MEML*).

Формат выводимого в поле *RESULT* результата по умолчанию совпадает со стандартным для Турбо Паскаля, но Вы можете управлять этим форматом с помощью специальных клавиш. Ключ помещается в конце выражения (переменной) и отделяется от него запятой. Например, запрос

```
MemL[$40:0],h
```

выдаст содержимое четырех байт оперативной памяти в шестнадцатеричном формате.

Допускаются следующие ключи:

C — формат *CHAR* (символьный);

S — формат *STRING* (строковый);

D — формат целых десятичных чисел;

\$, *H* или *X* — шестнадцатеричный формат;

Fl — формат *REAL* (вещественный); число *l* определяет количество значащих цифр в выводимом результате;

M — формат копии памяти: выводится содержимое переменной побайтно, начиная с младшего байта, в шестнадцатеричном формате; если ключ указан для выражения, он игнорируется;

P — формат *POINTER* (указатель); результат выводится в виде *Ptr(SEG, OFS)*, например, *Ptr(\$EA1, \$20)*;

R — формат *RECORD* (запись); выводится в круглых скобках список полей с указанием их значений, например: *(X:1; Y:10; Z:5)*.

Перед ключом формата можно указать целое число, которое трактуется как коэффициент повторения. Если, например, в поле *EXPRESSION* содержится

```
UserList[0], 4D
```

то в качестве результата будет выдано в формате целых десятичных чисел значение четырех последовательных элементов массива *UserList*, начиная с элемента 0, т.е. *UserList[0]*, *UserList[1]* и т.д. Коэффициент повторения относится только к переменным (по отношению к выражениям он игнорируется) и его можно указывать без ключа формата. Если, например,

```
var
```

```
UserList : array [0..20] of integer;
```

то обращения

```
UserList[0], 4D
```

```
UserList[0], 4
```

дадут идентичные результаты.

Выражения в поле *EXPRESSION* можно задавать многократно, можно использовать также ранее введенные выражения из протокола опции. Для выхода из диалога используйте клавишу *Esc* или поля *Cancel* и **■** — при работе с мышью. Опцию можно вызвать непосредственно из редактора командой *Ctrl-F4*. Отметим, что эта опция может использоваться как встроенный в Турбо Паскаль калькулятор.

ADD WATCH. С помощью этой опции можно указать отладчику те переменные и/или выражения, за изменением значений которых Вы хотели бы наблюдать при отладке программы. Указанные переменные и выражения вместе с их текущими значениями будут постоянно содержаться в окне наблюдения, доступ к которому возможен с помощью клавиши *F6*. Если Вы сделаете активным это окно, Вы сможете перемещаться в нем, вызывая при необходимости «прокрутку» его содержимого. Таким образом можно наблюдать за произвольным количеством переменных и выражений.

Опцию можно вызвать непосредственно из редактора командой *Ctrl-F7*. При этом сразу же все сказанное выше относительно вызова опции *DEBUG/EVALUATE/MODIFY*, т.е. выделение идентификатора, его редактирование и дополнение, использование коэффициента повторения и ключей формата. Сразу после добавления отслеживаемого выражения активируется окно наблюдения.

ADD BREAKPOINT. С помощью этой опции меню *DEBUG* Вы можете установить в текущей строке контрольную точку. Текущая строка — это строка с курсором в окне редактора. Если для нее установлена контрольная точка, строка выделяется цветом (яркостью). В программе можно установить произвольное количество контрольных точек. После запуска программы с установленными контрольными точками (точкой) отладчик прекратит исполнение программы перед выполнением того оператора, который содержится в первой (по логике работы программы) контрольной точке. При этом на экране появится окно редактора с контрольной точкой и среда перейдет к режиму отладки программы. Если контрольная точка задана для строки, не содержащей исполняемого оператора (например, для строки со словом *BEGIN*), программа остановится перед первым после этой строки исполняемым оператором. Останов в

контрольной точке можно сделать условным. Однажды установленная контрольная точка действует на каждое очередное обращение к соответствующей строке программы.

В диалоговом окне опции поле *Condition* задает условие останова. Это может быть произвольное логическое выражение с использованием любых переменных, констант, вызовов функций. Если к моменту исполнения оператора с контрольной точкой это выражение имеет значение *TRUE*, произойдет останов прогона и среда перейдет к режиму отладки. Поле *Pass count* указывает количество обращений к оператору с контрольной точкой, после которого произойдет останов.

С помощью команды *Ctrl-F8* контрольную точку можно установить/снять непосредственно из режима редактирования.

П1.2.8. Меню опции **TOOLS**

MESSAGES. Активизирует окно сообщений. Окно сообщений содержит вывод инструктивных программ типа *GREP* и позволяет использовать эти сообщения для поиска нужных фрагментов в текстах программ. Для поиска фрагмента подведите цветной указатель окна *Messages* к нужному сообщению и нажмите *Пробел* или дважды щелкните по этому сообщению мышью. Среда отыщет и покажет файл с нужным фрагментом текста программы.

GO TO NEXT. Ищет фрагмент, заданный следующим сообщением в окне *Messages*. Закрывает окно *Messages*, открывает нужный файл и позиционирует курсор на строку, соответствующую следующему по отношению к текущему (т.е. выделенному цветом) сообщению в окне. Опция вызывается непосредственно из окна редактора клавишами *Alt-F8*.

GO TO PREVIOUS. Ищет фрагмент, заданный предыдущим сообщением в окне *Messages*. Опция вызывается непосредственно из окна редактора клавишами *Alt-F7*.

GREP. Иницирует работу утилиты *GREP*. В строке *Enter program arguments* диалогового окна опции необходимо перечислить аргументы вызова *GREP*: имена процедур, функций, переменных, которые необходимо отыскать в текстовых файлах, а также имена этих файлов. По умолчанию параметром вызова *GREP* указывается **.PAS*, что означает поиск по всем *PAS*-файлам текущего каталога. Если к моменту вызова опции текстовый курсор стоял на некотором имени процедуры, функции или переменной, это имя появится в качестве аргумента вызова перед **.PAS*. После нажатия *Enter GREP* начнет поиск имени во всех файлах. В окне *Messages* появятся сообщения *GREP* с указанием имени файла, номера строки и фрагмента текста программы, где было найдено нужное имя. Опция вызывается непосредственно из окна редактора клавишами *Shift-F2*.

П1.2.9. Меню опции **OPTIONS**

COMPILER. Эта опция задает несколько параметров, с помощью которых Вы можете управлять генерацией машинного кода программы. Вид диалогового окна этой опции показан на рис.П1.8.

Опция *Force far calls* определяет генерацию машинного кода, рассчитанного на дальнюю модель памяти. В соответствии с архитектурой центрального процессора ПК могут использоваться две модели вызова процедур и функций: ближняя (*NEAR*) и дальняя (*FAR*). Ближняя модель обеспечивает адресацию в пределах текущего сегмента, дальняя используется для организации межсегментных связей. Если опция установлена в активное состояние, все вызовы процедур и функций будут использовать дальнюю (межсегментную) модель, в противном случае — ближнюю (внутрисегментную) модель. Ближняя модель дает более экономный код программы и выполняется быстрее, однако при организации оверлея и вызове из программы других программ с помощью процедуры *EXEC* нужно использовать дальнюю модель.

При активном состоянии опции *Overlays allowed* компилятор генерирует дополнительный код при компиляции оверлейных модулей. Этот код позволяет передавать строки и множества в качестве фактических параметров при обращении из одного оверлейного модуля в другой. Отметим, что Турбо Паскаль считает модуль оверлейным только в том случае, когда он откомпилирован с активной опцией *Overlays allowed*.

Опция *Word align data* определяет способ выравнивания переменных и констант в памяти: если опция активна, каждая переменная и константа начинается в байте с четным адресом, т.е. выравнивается на начало машинного слова, если неактивна, переменные и константы располагаются в памяти сплошной цепочкой. Выравнивание по словам увеличивает скорость выполнения программ ценой несколько неэкономного расходования памяти.

Активное состояние опции *286 instructions* предписывает компилятору создавать код программы с полным набором команд микропроцессора Intel 80286. В неактивном состоянии опции компилятор порождает код, соответствующий набору команд микропроцессора Intel 8088 и представляющий собой подмножество команд микропроцессора Intel 80286. В целях переносимости программ имеет смысл устанавливать неактивное состояние этой опции, так как в процессе счета программа не проверяет фактическое наличие микропроцессора Intel 80286 и не может эмулировать его систему команд.

[!] Compiler Options	
Code generation	
☐ Force far calls	☐ Word align data
☐ Overlays allowed	☒ 286 instructions
Runtime errors	Syntax options
☒ Range checking	☐ Strict var-strings
☒ Stack checking	☐ Complete boolean eval
☒ I/O checking	☐ Extended syntax
Numeric processing	Debugging
☒ 8087/80287	☒ Debug information
☐ Emulation	☒ Local symbols
Conditional defines	
OK Cancel Help	

Рис.П1.8. Диалоговое окно опции Options/Compiler

В активном состоянии опции *Range checking* генерируется дополнительный код для проверки возможного выхода значений переменных за границы диапазона. Программа, откомпилированная с активной опцией, занимает несколько больший объем памяти и исполняется медленнее, зато дает возможность контролировать выход индексов за пределы, определенные в описании массивов.

Опция *Stack checking* аналогична опции *Range checking* и контролирует возможное переполнение программного стека.

Опция *I/O checking* используется для включения/отключения генерации программных кодов, контролирующих правильность операций ввода-вывода.

Установка в неактивное состояние опции *Strict var-strings* позволяет отказаться от проверки на совпадение длины формального и фактического параметра-строки при обращении к процедуре или функции. Если установлено активное состояние этой опции, компилятор вставляет в программу команды для сравнения длины строк.

В активном состоянии опции *Complete boolean eval* все логические выражения вычисляются в программе полностью, в неактивном состоянии вычисление прекращается в тот момент, когда становится ясен окончательный результат. Допустим имеется такой фрагмент программы:

```
.....
Function MyFunc(var x : integer) : Boolean
begin
  x := x+1;
```

```

MyFunc := x>10
end;
.....
x := 0;
if False and MyFunc(x) then x := 10;
.....

```

После его компиляции с неактивной опцией *Complete boolean eval* исполнение этого фрагмента даст $X=0$, так как не произойдет обращения к функции *MYFUNC*: выражение *FALSE AND MYFUNC* всегда имеет значение *FALSE* вне зависимости от того, что является вторым операндом операции *AND*. Если же к моменту компиляции программы было установлено активное состояние этой опции, вычисление логического выражения продолжится до конца, состоится вызов функции *MYFUNC* и переменная *X* получит значение 1. Разумеется, и в том и в другом случае не будет исполняться оператор $x := 10$.

Активное состояние опции *Extended syntax* дает возможность использовать в программе расширенный синтаксис Турбо Паскаля, который разрешает вызывать определенные программистом функции не только в выражениях, но и в виде отдельного оператора, подобно вызову процедуры. Отметим, что эта опция не распространяется на стандартные функции.

Опция *8087/80287* ориентирует компилятор на работу с арифметическим сопроцессором. При неактивном состоянии все операции с вещественными данными реализуются программно и в программе разрешается использовать только один вещественный тип *REAL*. Если опция установлена в активное состояние, компилятор будет создавать код, содержащий обращения к числовому сопроцессору, причем программе становятся доступны также типы *SINGLE*, *DOUBLE*, *EXTENDED* и *COMP* (см. гл.4).

Опция *Emulation* указывает компилятору, надо ли создавать такой код программы, который будет одинаково пригоден при работе на ПК с арифметическим сопроцессором или без него. Программа сама определит наличие сопроцессора и, если он имеется, будет использовать все его возможности; если же сопроцессора нет, его работа будет эмулироваться программно. В этом случае программе становятся доступны все вещественные типы. Активное состояние этой опции увеличивает размеры программы за счет подключения процедур эмуляции, но делает ее независимой от аппаратных особенностей ПК. Отметим, что опция *Emulation* игнорируется, если неактивна опция *8087/80287*.

Активное состояние опции *Debug information* устанавливает режим генерации отладочной информации в процессе компиляции программы. Отладочная информация представляет собой специальные таблицы, позволяющие установить однозначную связь между операторами исходного текста программы и теми кодами, которые порождает компилятор. Только после компиляции с активной опцией *Debug information* становится возможной автоматическая локализация ошибки периода исполнения, а также пошаговая отладка программы. Активное состояние опции увеличивает размер *TPU*-файлов и объем оперативной памяти, занимаемой программой, если она работает под управлением среды Турбо Паскаля, но не влияет на размер той же программы, запускаемой вне среды под управлением ДОС. Иными словами, отладочные таблицы отладки загружаются в память только средой Турбо Паскаля, а ДОС игнорирует эту информацию.

Опция *Local symbols* аналогична опции *Debug information* и относится к именам локальных и глобальных переменных: если опция установлена в активное состояние, среда получит возможность доступа на этапе отладки к переменным по их именам.

В поле *Conditional defines* Вы можете задать условия, которые используются в операторах условной компиляции (см. дальше п.П1.3).

MEMORY SIZES. В диалоговом окне опции *OPTIONS/MEMORY SIZES* используются три поля ввода. С их помощью можно регулировать размеры памяти, которую занимает работающая программа:

Stack size — размер программного стека; по умолчанию 16384 байта, максимум — 65535 байт;

Low heap limit — минимальный размер кучи; по умолчанию 0;

High heap limit — максимальный размер кучи; по умолчанию 655360 байт; этот параметр не может быть меньше параметра *Low heap limit*.

Для оценки необходимых программе объемов памяти следует учесть, что все локальные переменные при каждом обращении к процедуре (функции) размещаются в стеке, а при выходе из нее стек освобождается. Таким образом, требуемый размер стека определяется количеством вложенных вызовов процедур (функций) и суммарным количеством их ло —

кальных переменных. Величина кучи определяется реальными потребностями программы в динамической памяти. Если установлен максимально возможный размер кучи 655360 байт, то такая программа после загрузки займет всю доступную оперативную память, а это исключит возможность запуска из нее других программ.

LINKER. В диалоговом окне этой опции имеются две группы переключаемых опций, с помощью которых регулируется режим работы компоновщика Турбо Паскаля: опции группы *Map file* управляют выходным документом компоновщика, опции группы *Link buffer* — использованием памяти. Выходной документ компоновщика (карта распределения памяти) бывает полезен при отладке программы с помощью внешнего отладчика. Опция *Off* запрещает формирование карты. Опция *Segments* формирует сегментную карту с указанием адреса запуска программы и сообщениями об ошибках периода компоновки программы. Опция *Public* дает такую же карту, как и опция *Segments*, и дополнительно приводит список внешних символов в алфавитном порядке. Наконец, опция *Detailed* дает полную карту распределения памяти. Опция *Memory* предписывает компоновщику использовать оперативную память для размещения своих таблиц и временного хранения компоновочной программы, при активной опции *Disk* компоновщик для этих целей использует пространство диска. Если активна опция *Memory*, компоновщик будет работать значительно быстрее, однако при разработке крупных программ ему может не хватить оперативной памяти и он не скомпилирует программу.

Вообще, следует помнить о том, что даже довольно большой объем оперативной памяти ПК (640 Кбайт) может оказаться недостаточным для разработки с помощью среды Турбо Паскаля крупных программных проектов: ведь сам Турбо Паскаль занимает в памяти 304 Кбайт. Если обнаружена нехватка памяти, среда дает сообщение

Out of memory
(не хватает памяти)

и устанавливает курсор в конец программы. В этом случае следует прежде всего попытаться сэкономить память за счет установки в активное состояние опции *Disk*¹. Еще примерно 44 Кбайта памяти можно сэкономить за счет отказа от автоматической загрузки системной библиотеки *TURBO.TPL* (см. ниже опцию *OPTIONS/ENVIRONMENT/STARTUP*). Наконец, может оказаться необходимым отказ от услуг самой среды Турбо Паскаля на этапе прогона программы. Для этого нужно установить опцию *COMPILE/DESTINATION* в состояние *DISK*, создать программу с помощью опций *MAKE* или *BUILD*, выйти из среды и запустить программу. В этом случае программа получает в свое распоряжение всю память ПК, но Вы лишаетесь возможности отлаживать ее средствами встроенного отладчика. В некоторых случаях за счет оверлейной структуры программы (см. гл. 11) ее размеры удается уменьшить настолько, что даже крупная программа помещается в памяти вместе со средой. Если, несмотря на все меры экономии, памяти все-таки не хватает, можно полностью отказаться от услуг среды и использовать автономный компилятор — компоновщик *TPC.EXE*.

DEBUGGER. Эта опция определяет используемый отладчик и режим обновления экрана дисплея в процессе отладки. Если активна опция *Integrated*, к программе будет добавлена информация, необходимая для работы встроенного отладчика. Только в этом состоянии опции можно использовать контрольные точки и пошаговую отладку. При активизации опции *Standalone* к *EXE*-файлу программы будут добавлены соответствующие таблицы, которые позволяют вести отладку программы вне среды Турбо Паскаля с помощью внешнего отладчика *TD.EXE*. Три других опции сообщают среде, в каких случаях следует переключать экран с воспроизведения окна редактора на окно программы. В режиме *Smart* среда будет переключать экран по мере надобности — только если в очередном операторе программы было обращение к экрану для вывода или к клавиатуре для ввода. Переключение на окно программы будет также и тогда, когда отладчик «перескакивает» через вызов процедуры (функции) по клавише *F8*, но в этой процедуре (функции) есть обращение к экрану. Если установлен режим *Always*, переключение будет происходить перед исполнением любого оператора программы. Наконец, в режиме *None* среда никогда не переключает экран, даже

¹ Турбо Паскаль 7.0 включен в комплект поставки более мощной системы программирования Borland Pascal with objects 7.0. Входящая в эту систему среда программирования BP использует расширенную память для размещения таблиц компилятора и компоновщика. В то же время язык Турбо Паскаля представляет собой подмножество языка BP, что дает возможность компилировать программы Турбо Паскаля без какой-либо их переделки в среде BP. Используйте, если это возможно, среду BP для разработки крупных программ.

если он требуется для вывода данных, т.е. вывод программы будет накладываться на текст программы. Испорченный в результате такого прогона текст в окне редактора можно обнови́ть с помощью опции *Window/Refresh display*.

DIRECTORIES. Четыре поля ввода в диалоговом окне опции *OPTIONS /DIRECTORIES* позволяют определить четыре группы функциональных каталогов Турбо Паскаля.

EXE & TPU directories указывает тот каталог, в который будут помещаться готовые к работе программы в виде *EXE*—файлов и результат компиляции модулей в виде *TPU*—файлов. Если каталог не указан, эти файлы будут помещаться в текущий каталог — именно такое состояние этой опции соответствует стандартной настройке среды. Не рекомендуется устанавливать в этой опции каталог, содержащий файлы системы Турбо Паскаль.

Include directories — здесь следует перечислить те каталоги, в которых Турбо Паскаль будет искать включаемые файлы, т.е. файлы, задаваемые директивой компилятору *{ \$I <имя файла> }*. При указании нескольких каталогов, они перечисляются через точку с запятой. Отметим, что поиск в этих каталогах идет только в том случае, если включаемый файл не найден в текущем каталоге.

Unit directories — задает каталоги, в которых среда ищет *TPU*—файлы, если они не обнаружены в текущем каталоге. В этой опции обычно указывается каталог, содержащий файл *GRAPH.TPU* (если в программе используются графические средства Турбо Паскаля), а также каталог, указанный в поле *EXE & TPU directories*. При перечислении нескольких каталогов они разделяются точкой с запятой.

Если в своей программе Вы используете внешние процедуры и функции (см. гл. 11), они должны быть представлены в виде *OBJ*—файлов. Поле *Object directories* задает один или несколько каталогов, в которых Турбо Паскаль будет искать эти файлы, если их нет в текущем каталоге.

ENVIRONMENT. При вызове этой опции разворачивается еще одно дополнительное меню, содержащее пять опций. Эти опции описываются ниже.

ENVIRONMENT/Preferences. Диалоговое окно этой опции показано на рис.П1.9.

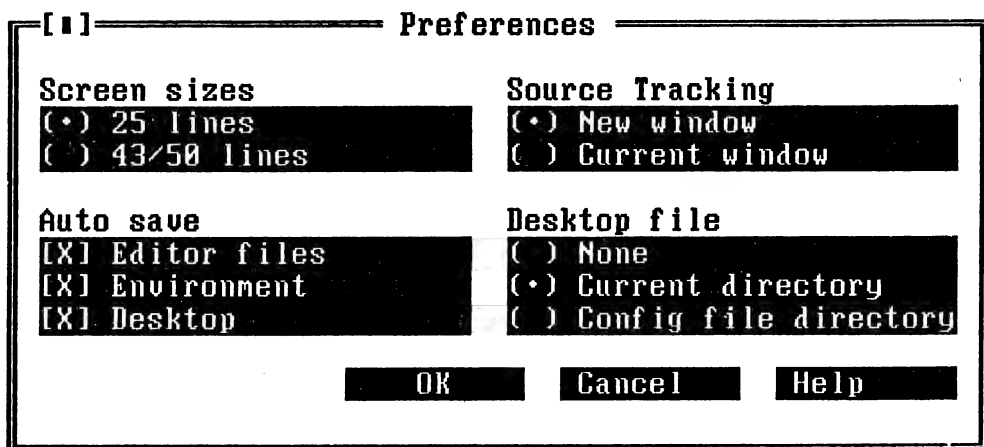


Рис.П1.9. Диалоговое окно опции *Options/Environment/Preferences*

Опции группы *Screen sizes* определяют размер текстового экрана: если активна опция *25 lines*, на экране будет 25 строк, если активна опция *43/50 lines*, на экране будет 43 или 50 строк в зависимости от того, оснащен ли Ваш ПК дисплеем с адаптером *EGA* или *VGA*. Опции *Source tracking* определяют способ использования текущего окна редактора в процессе отладки. Если активна опция *New window*, прослеживаемая программа будет загружаться в новое окно редактора (если, разумеется, она еще не загружена в одно из ранее открытых окон), если активна опция *Current window*, — в текущее окно.

Группа опций *Auto save* регулирует запись на диск текущей программы и информации о ней. Опция *Editor files* предписывает автоматически сохранять на диске содержимое всех окон редактора перед прогоном программы, если только текст в окне изменялся после последней записи на диск. Чрезвычайно полезная опция, которую я настоятельно рекомендую всегда

устанавливать в активное состояние — автосохранение измененного текста программы избавит Вас от многих неприятностей при работе с плохо отлаженной программой или на ПК с ненадежной (сбойной) памятью. Разумеется, дополнительное обращение к диску затягивает переход к прогону/отладке, однако Вы по достоинству оцените эту услугу среды после первого же «зависания» программы. Опция *Environment* задает режим автоматического сохранения текущей настройки среды Турбо Паскаля в файле конфигурации *TURBO.TP*. Настройка среды будет сохранена автоматически при временном или окончательном выходе из Турбо Паскаля, если она изменилась с момента последней записи в этот файл. Активное состояние опции *Desktop* требует от среды автоматически сохранять на диске в файле *TURBO.DSK* информацию об открытых окнах, их размерах, контрольных точках и т.п. Использование этой опции позволит при очередной загрузке Турбо Паскаля автоматически получить точную копию того состояния среды, в котором Вы покинули ее в последний раз. Следует учесть, что опция *Desktop* игнорируется, если не был создан конфигурационный файл *TURBO.TP*.

Группа опций *Desktop file* уточняет режим создания файла *TURBO.DSK*: если выбрана опция *None*, файл не будет создаваться, даже если опция *Desktop* активна; опция *Current directory* определяет размещение этого файла в текущем каталоге, а опция *Config file directory* — в том же каталоге, где размещается конфигурационный файл *TURBO.TP*.

ENVIRONMENT/Editor. Диалоговое окно этой опции показано на рис.П1.10.

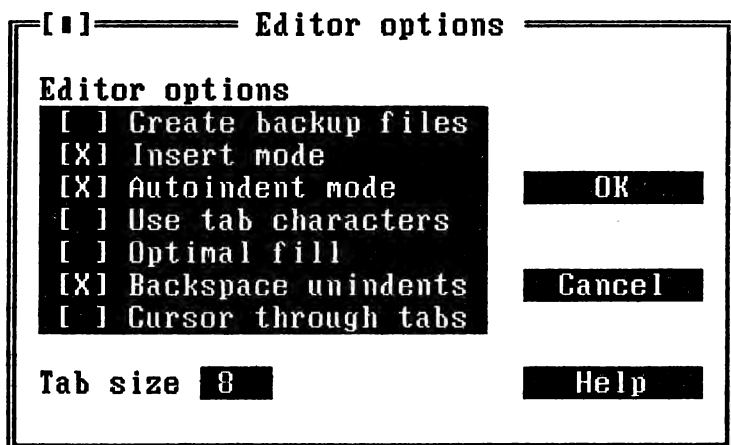


Рис.П1.10. Диалоговое окно опции *Options/Environment/Editor*

Активное состояние опции *Create backup files* заставит среду при записи файла с текстом программы проверить, существует ли уже одноименный файл; если существует, он будет переименован в файл с тем же именем и расширением *.BAK*, что позволит иметь на диске предыдущую версию только что сохраненной информации. Опция *Insert mode* указывает на основной режим работы редактора: если она активна, редактор работает в режиме вставки, неактивна — в режиме замены. Опция определяет лишь начальный режим работы сразу после загрузки Турбо Паскаля, так как в процессе работы с редактором Вы всегда можете переключить режим клавишей *Ins*. Активизация опции *Autoindent mode* облегчит Вам выделение отступами условных и составных операторов, так как в этом случае каждая новая строка будет начинаться с таким же отступом от левого края экрана, что и предыдущая строка. Опция *Use tab characters* определяет использование символов табуляции: если она активна, нажатие на клавишу табуляции вставит в текст специальный символ табуляции, если неактивна, в текст будет вставлено нужное число пробелов. Дополнительно к этому активная опция *Optimal fill* заставит редактор вставлять минимально возможное число символов табуляции и пробелов при нестандартной длине табуляционного интервала (не 8 символов). Активная опция *Backspace unindents* связывает с клавишей *Backspace* функции уничтожения предыдущего символа с учетом автоотступа: если слева от курсора нет ни одного значащего символа, нажатие на *Backspace* сдвинет курсор к позиции предыдущего автоотступа. Обычно эта опция устанавливается в активное состояние одновременно с опцией *Autoindent mode*. Опция *Cursor through tabs* определяет перемещение курсора по строке при нажатии клавиши смещения

курсор влево или вправо: если она активна, курсор смещается скачком всякий раз, когда в строке встречается символ табуляции; при неактивной опции он всегда смещается только на одну позицию. В поле *Tab size* Вы можете задать длину табуляционного интервала.

ENVIRONMENT/Mouse.

Группа опций *Right mouse button* определяет способ использования правой кнопки мыши в сочетании с нажатой и удерживаемой клавишей *Ctrl*:

Опция	Функция правой кнопки
Nothing	Нажатие на кнопку игнорируется
Topic search	Эквивалент действию опции Help/Topic search
Go to cursor	Эквивалент действию опции Run/Go to cursor
Breakpoint	Включает/отключает контрольную точку
Evaluate	Эквивалент действию опции Debug/Evaluate
Add watch	Эквивалент действию опции Debug/Add watches

Например, если активной сделана опция *Go to cursor*, то при работе с мышью нажатие на ее правую кнопку при нажатой и удерживаемой клавише *Ctrl* будет эквивалентно вызову опции *Run/Go to cursor* или нажатию на клавишу *F4*.

При обращении к справочной службе в тексте могут встретиться ссылки на другие разделы справочных сообщений. Эти ссылки выделяются в справочном сообщении цветом (в стандартной настройке это — яркий желтый цвет). Соответствующие справки можно вызвать с помощью мыши: нужно установить указатель на ссылку и дважды подряд с небольшим промежутком времени (доли секунды) нажать левую кнопку легкими отрывистыми ударами. Этот прием требует определенной сноровки, так как слишком большая пауза между нажатиями воспринимается средой как два нажатия на левую кнопку, а не как команда вызова соответствующей справки. Поле *Mouse double click* в диалоговом окне опции *Environment/Mouse* используется для регулирования промежутка времени, в течение которого два нажатия воспринимаются как одна команда: для увеличения интервала установите указатель мыши ближе к правой границе поля и нажмите левую кнопку.

Переключаемая опция *Reverse mouse buttons* используется для реверсирования функции кнопок мыши: при ее активизации нажатие на левую кнопку воспринимается средой как нажатие на правую кнопку и наоборот. Эта опция может быть полезна в том случае, если Вам удобнее работать с мышью левой рукой.

ENVIRONMENT/Startup. На рис.П1.11 показан вид диалогового окна этой опции.

С помощью опции *Dual monitor support* можно сообщить среде, что Ваш ПК оснащен двумя дисплеями. После активизации этой опции на дополнительный дисплей будет выводиться окно прогона программы.

Опция *Graphics screen save* обеспечивает сохранение в оперативной памяти ПК копии графического экрана. Эту опцию следует активизировать, если Вы разрабатываете (и отлаживаете в среде Турбо Паскаля) программу, использующую графические средства ПК. При активизации этой опции Вы всегда сможете увидеть в окне прогона программы те графические изображения, которые она формирует.

Опция *EGA/VGA palette save* позволяет сохранить в оперативной памяти цветовую палитру экрана, поэтому изменения этой палитры в отлаживаемой программе не будут влиять на вид окон среды Турбо Паскаль.

Отметим, что обе предыдущие опции используют часть оперативной памяти для сохранения графического экрана и цветовой палитры, поэтому их следует отключать, если Ваша программа критична к объему доступной памяти и/или не использует графические средства ПК и не меняет цветовую палитру.

Опция *CGA snow checking* может быть неактивна, если Ваш ПК оснащен адаптерами типа EGA или VGA. Но если в нем используется адаптер типа CGA, неактивное состояние этой опции будет создавать помехи на экране в виде «снега» при смене на нем изображения (активное состояние этой опции несколько замедляет темп обновления информации на экране, но зато гарантирует отсутствие «снега» на дисплеях любого типа).

[]
Startup options

☐ Dual monitor support
☒ Graphics screen save
☒ EGA/UGA palette save
☐ CGA snow checking
☐ LCD color set
☒ Use expanded memory
☒ Load TURBO.TPL

Window heap size
32

Editor heap size
28

Overlay heap size
112

Swap file directory

OK

Cancel

Help

Рис.П1.11. Диалоговое окно опции Options/Environment/Startup

Опцию *LCD color set* следует активизировать только в том случае, когда ПК оснащен жидкокристаллическим дисплеем (обычно такие дисплеи устанавливаются на переносимых ПК).

На машинах класса IBM PC/XT может устанавливаться отображаемая память типа EMS (на современных ПК с процессорами 80386 и выше эта память может эмулироваться). Активизация опции *Use expanded memory* укажет среде на возможность использования этой памяти для размещения оверлейных модулей, сохранения копий графического экрана и некоторых других функций (см. ниже).

При стандартной настройке среды сразу после загрузки Турбо Паскаля в оперативную память считывается системная библиотека *SYSTEM.TPU* из библиотечного файла *TURBO.TPL*. Вы можете отказаться от этого (и таким образом сэкономить для программы около 44 Кбайт), если сделаете неактивной опцию *Load TURBO.TPL*. Следует учесть, что в этом случае системная библиотека должна быть выделена в виде отдельного файла *SYSTEM.TPU* в каталоге, путь к которому указан в поле *Unit directories* опции *Options/Environment/Directories*. Извлечение модуля *SYSTEM.TPU* из библиотеки *TURBO.TPL* осуществляется с помощью утилиты *TPUMOVER.EXE*, входящей в комплект поставки системы Турбо Паскаль. Для этого используется команда ДОС вида:

```
TPUMOVER TURBO.TPL *SYSTEM.TPU
```

(перед именами утилиты, библиотеки и системного модуля можно указывать необходимые пути; символ * перед именем системного модуля определяет операцию извлечения модуля). Если программа использует другие библиотеки, входящие в *TURBO.TPL* (*CRT*, *DOS*, *PRINTER*, *OVERLAY* — см. гл.10), их также нужно выделить в отдельные *TPU*-файлы.

В поле *Window heap size* задается размер оперативной памяти (в килобайтах), который выделяется для хранения неактивных окон среды, в поле *Editor heap size* — для хранения содержимого окон редактора, а в поле *Overlay heap size* — для хранения оверлейных модулей. Если активизирована опция *Use expanded memory*, эта память выделяется из дополнительной памяти ПК, если неактивна — из основной. Соответствующей установкой значений в этих полях можно регулировать размеры буферной памяти в следующих пределах:

Поле	Минимум	Максимум
Window heap size	24 Кбайт	64 Кбайт
Editor heap size	28 Кбайт	128 Кбайт
Overlay heap size	64 Кбайт	256 Кбайт

В поле *Swap file directory* можно указать «быстрый» диск, с которым среда будет производить динамический обмен данными (свопинг) в процессе своей работы. В качестве

«быстрого» диска обычно указывается виртуальный диск, т.е. участок оперативной памяти, который используется подобно механическому диску (виртуальный диск организуется средствами ДОС). Если в этом поле ничего не указано, для свопинга используется текущий каталог.

ENVIRONMENT/Colors. С помощью диалогового окна этой опции Вы можете установить нужную цветовую палитру отдельных элементов среды Турбо Паскаля. В колонке *Group* указаны следующие элементы среды:

<i>Desktop</i>	вид экрана;	<i>Watches</i>	окно отладки;
<i>Menus</i>	меню;	<i>Call stack</i>	окно стека;
<i>Dialogs</i>	диалоговые окна;	<i>Register</i>	окно регистров;
<i>Editor</i>	окна редактора;	<i>Output</i>	окно программы;
<i>Help</i>	окно помощи;	<i>Compiler</i>	окно компилятора.

В колонке *Item* детализируются эти элементы (например, цвет рамки окна, цвет основных символов, цвет выделения и т.п.). В поле *Foreground* задается цвет символов, а в поле *Background* — цвет фона.

OPTIONS/Open. Здесь Вы можете указать имя конфигурационного файла, из которого среда должна получить информацию о своей настройке.

OPTIONS/Save. Сохраняет текущую настройку среды в конфигурационном файле.

OPTIONS/Save as. С помощью этой опции из меню *Options* можно указать каталог и файл, в котором среда будет сохранять свою настройку (по умолчанию это файл *TURBO.TP*).

П1.2.9. Меню опции WINDOW

TILE. Располагает окна так, чтобы каждое было видно на экране и все они имели бы приблизительно одинаковые размеры..

CASCADE. Располагает на экране окна редактора таким образом, чтобы были видны рамки каждого из них. Используется для организации более удобной работы с помощью мыши.

CLOSE ALL. Закрывает все открытые окна.

REFRESH DISPLAY. Удаляет следы вывода программы, работавшей в режиме отладки с установленной опцией *Options/Debugger/Display swapping/None*.

SIZE/MOVE. Эта опция обеспечивает перемещение окна по экрану и/или изменение его размеров. Вызывается из редактора командой *Ctrl-F5*.

ZOOM. Распахивает активное окно на весь экран или возвращает ему прежний вид. Вызов из редактора клавишей *F5*.

NEXT. Активизирует очередное окно. Вызывается из редактора клавишей *F6*.

PREVIOUS. Активизирует предыдущее активное окно. Вызывается из редактора командой *Shift-F6*.

CLOSE. Закрывает активное окно. Вызывается из редактора командой *Alt-F3*.

LIST. Выводит на экран список всех открытых окон среды. Вызывается из редактора командой *Alt-0*.

П1.2.10. Меню опции HELP

CONTENTS. Выводит на экран содержание справочной службы.

INDEX. Выводит на экран алфавитный список всех ссылок справочной службы. Вызывается из редактора командой *Shift-F1*.

TOPIC SEARCH. Осуществляет поиск в окрестности курсора зарезервированного слова или имени стандартной процедуры (функции) и дает соответствующую справку. Вызывается из редактора командой *Ctrl-F1*.

PREVIOUS TOPIC. Выводит на экран предыдущее справочное сообщение. Вызывается из редактора командой *Alt-F1*.

HELP ON HELP. Дает справку о том, как пользоваться справочной службой. Отметим, что в сообщениях справочной службы все перекрестные ссылки выделяются цветом. Вы можете подвести к любой из них указатель мыши и двойным нажатием на ее левую кнопку вызвать на экран соответствующее справочное сообщение (или сместить к ней указатель с помощью клавиш перевода курсора и нажать *Enter*).

FILES. С помощью этой опции Вы можете установить нужные файлы справочной службы.

COMPILER DIRECTIVES. Показывает справку о директивах компилятора.

RESERVED WORDS. Показывает справку о зарезервированных словах.

STANDARD UNITS. Показывает справку о стандартных модулях.

TURBO PASCAL LANGUAGE. Показывает справку о языке Турбо Паскаль.

ERROR MESSAGES. Показывает справку о сообщениях об ошибках.

ABOUT. Выводит информацию о авторских правах и версии Турбо Паскаля.

П1.3. ДИРЕКТИВЫ КОМПИЛЯТОРА

В меню *OPTIONS/COMPILER* включены опции, с помощью которых можно управлять работой компилятора. В ряде случаев бывает необходимо временно отменить действие той или иной опции при трансляции некоторого фрагмента программы. Особенно часто, например, такая необходимость возникает при обращении к диску: если программа пытается прочитать несуществующий файл или записать данные на защищенный диск, возникнет ошибка периода исполнения и программа аварийно закончит свою работу. В то же время, если отключить опцию *I/O CHECKING*, этого не произойдет, программа сможет проанализировать последствия обращения к диску и предпринять альтернативные действия.

В Турбо Паскале можно использовать *директивы компилятора*, которые в виде особым образом оформленных комментариев вставляются в текст программы и модифицируют те или иные возможности компилятора в процессе компиляции. Директивы могут быть переключающими, условными и параметрическими. Переключающие директивы воздействуют на те опции, которые включены в диалоговое окно *OPTIONS/COMPILER*; условные директивы определяют условия, при которых компилируются те или иные фрагменты программы; параметрические директивы задают параметры, которые должен учитывать компилятор.

Все директивы оформляются в виде особых комментариев: они обрамляются фигурными скобками, а за открывающей скобкой должен без пробелов следовать знак доллара (десятичный код 36). Как только в процессе разбора исходного текста программы компилятор встретит такого рода последовательность символов, он воспримет их как директиву и нужным образом изменит свою работу.

Переключающая директива содержит букву, обозначающую опцию, и знак «+» или «-». Знак «+» означает установку опции в активное состояние, знак «-» — в пассивное состояние. Например, директива *{\$I-}* означает временное отключение контроля ошибок ввода-вывода, директива *{\$R+}* — включение контроля границ диапазона. В одной директиве можно перечислить несколько опций, например:

{\$A+,B-,D+,E+,F+,I+,L+,N+,O-,R+,S+,V+}

Следует учесть, что директивы компилятора действуют от момента своего появления в тексте до конца текущего модуля, т.е. локализируются в теле модуля, в то время как опции, установленные в самой среде, распространяются на все модули и основную программу. В случае конфликта между директивами и опциями, предпочтение отдается директивам. Таким образом, правильно расставленные директивы обеспечивают нужную компиляцию программы независимо от настройки среды. Они особенно полезны в случае, когда компиляция осуществляется автономным компилятором *TP.EXE*.

Некоторые директивы компилятора могут действовать только на часть текста программы, такие директивы называются *локальными*; в отличие от этого *глобальные* директивы располагаются в самом начале текста программы (модуля) и действуют сразу на всю программу (модуль) в целом.

Ниже приводится список всех директив компилятора. В скобках дается действие директив для знака «-». Знаком * отмечены локальные директивы.

`{A+}` — выравнивать данные на границу слова (байта);
`{B+}` — вычислять логические выражения полностью (до получения результата);
`{D+}` — разрешить (запретить) работу со встроенным отладчиком;
`{E+}` — включить (отключить) режим программной эмуляции сопроцессора;
`{F+}` — использовать дальнюю (ближнюю) модель вызова;
`{G+}` — использовать (не использовать) полный набор команд микропроцессора Intel 80286 (микропроцессора Intel 80386);
`{I+}` — включить (отключить) контроль операций ввода — вывода;
`{L+}` — включить (не включать) локальные символы в информацию для отладчика;
`{N+}` — использовать числовой сопроцессор (реализовать операции с плавающей точкой программно);
`{O+}` — разрешить (не разрешать) создание оверлейной структуры;
`{R+}` — включить (отключить) контроль границ диапазона;
`{S+}` — включить (отключить) контроль возможного переполнения стека;
`{V+}` — включить (отключить) контроль длины строк при обращении к процедуре или функции;
`{X+}` — использовать (не использовать) расширенный синтаксис.

К условным директивам относятся следующие локальные директивы компилятора:

`{DEFINE <условный символ>}` — установить условный символ;
`{IFDEF <условный символ>}` — проверить установку условного символа; если символ установлен, будет компилироваться вся следующая за директивой часть программы вплоть до директив `{ELSE}` или `{ENDIF}`, в противном случае этот фрагмент программы будет пропущен компилятором;
`{IFNDEF <условный символ>}` — проверить установку локального символа; обратна директиве `{IFDEF}`, т.е. действует только в том случае, когда условный символ не установлен;
`{ELSE}` — определяет начало альтернативного фрагмента программы; этот фрагмент будет компилироваться в том случае, если условный символ, проверенный предыдущей по тексту программы директивой `{IFDEF}` или `{IFNDEF}`, не установлен; альтернативная часть вместе с директивой `{ELSE}` может опускаться;
`{ENDIF}` — ограничивает область действия директив `{IFDEF}`, `{IFNDEF}` или `{ELSE}`.

Установить условие — это значит с помощью опции `OPTIONS/COMPILER/CONDITIONAL DEFINES` среды Турбо Паскаля или с помощью директивы компилятора `{DEFINE}` ввести некоторое слово (условный символ), которое затем будет управлять компиляцией какого-либо фрагмента программы. Если, например, использовать в качестве условного символа слово `DEBUG`, то можно написать такой фрагмент программы:

```

{$IFDEF Debug}
  WriteLn ('Отладка: x=', x);
{$ENDIF}

```

Теперь, если установлен (задан в среде или введен с помощью директивы `{DEFINE}`) условный символ `DEBUG`, в программе будет откомпилирован оператор `WRITELN`, если этот условный символ не задан, компилятор пропустит этот оператор.

Отметим, что условные символы никак не связаны с множеством идентификаторов самой программы и недоступны ей на этапе исполнения. Для предыдущего примера нельзя написать

```
WriteLn (Debug);
```

если, разумеется, в программе не определена переменная или константа с этим именем.

Одновременно в программе может быть установлено сколько угодно условных символов. Для тестирования нескольких условий используется вложение условных директив компилятора, например:

```

{$IFDEF Var1}
{$IFDEF Var2}
  a := n;
{$ELSE}
  a := 0;
{$ENDIF}
{$ENDIF}

```


В директивах `{$IFDEF}` или `{$IFNDEF}` программист может использовать следующие стандартные условные символы, которые устанавливаются в среде автоматически:

`VER70` — указывает версию 7.0 Турбо Паскаля (для других версий используются другие условные символы: `VER40`, `VER50`, `VER55`, `VER60`);

`MSDOS` — указывает на то, что программа будет исполняться под управлением операционной системы MS-DOS (PC DOS); версии Турбо Паскаля для других операционных систем не будут устанавливать этот символ;

`CPU86` — указывает на то, что программа будет исполняться на ПК с микропроцессором, принадлежащим семейству Intel 80x86;

`CPU87` — указывает на то, что в составе аппаратных средств ПК обнаружен (к моменту компиляции программы!) арифметический сопроцессор семейства Intel 80x87.

К параметрическим относятся директивы:

`{$M <стек>, <нижн>, <верх>}` — установить требуемые размеры памяти (глобальная директива); здесь `<стек>` — размер стека; `<нижн>` — нижняя граница динамической памяти; `<верх>` — верхняя граница динамической памяти;

`{$I <имя файла>}` — включение `PAS`-файла: `<имя файла>` — имя включаемого файла;

`{$L <имя файла>}` — включение `OBJ`-файла: `<имя файла>` — имя включаемого файла.

В директиве `{$M}` все размеры задаются в байтах, следуют друг за другом в указанной последовательности и отделяются запятыми. Между буквой `M` и первой цифрой размера стека должен быть хотя бы один пробел, между последней цифрой верхней границы динамической памяти и закрывающей фигурной скобкой не должно быть никаких символов, например:

`{$M 16384, 0, 655360}`

Включаемый файл, определенный в директиве `{$I}`, должен содержать исходный текст фрагмента программы. Компилятор использует эту директиву как указание обратиться к дисковому файлу за очередной строкой программы. После того, как весь фрагмент программы из файла будет прочитан, компилятор продолжит чтение строк из файла редактора. Если в имени файла опущено расширение, используется стандартное расширение `.PAS`. Диск и каталог, в котором находится файл, задаются опцией `OPTIONS/ENVIRONMENT/DIRECTORIES`.

Директива `{$L}` предназначена для указания компилятору файла, в котором содержится результат трансляции ассемблерной процедуры или функции, объявленной в программе как внешняя (`EXTERNAL`, см. гл. 11). Если в имени файла опущено расширение, используется стандартное расширение `.OBJ`. Диск и каталог, в котором находится файл, задаются опцией `OPTIONS/ENVIRONMENT/DIRECTORIES`.

П1.4. РЕДАКТОР

Основные приемы работы с редактором описаны в гл. 1. Ниже приводится полное описание команд редактора. Все команды редактора можно разделить на команды перемещения курсора, команды удаления/вставки, команды работы с блоками, прочие.

При их описании используются следующие обозначения клавиш управления курсором:

<code>ВЛ</code>	курсор влево;	<code>ВВ</code>	курсор вверх;
<code>ВП</code>	курсор вправо;	<code>ВН</code>	курсор вниз.

П1.4.1. Команды перемещения курсора

`Ctrl-S` или `ВЛ` — на символ влево;

`Ctrl-D` или `ВП` — на символ вправо;

`Ctrl-A` или `Ctrl-ВЛ` — на слово влево;

`Ctrl-F` или `Ctrl-ВП` — на слово вправо;

`Ctrl-E` или `ВВ` — на строку вверх;

`Ctrl-X` или `ВН` — на строку вниз;

`Ctrl-W` — прокрутка вниз на строку вместе с курсором;

`Ctrl-Z` — прокрутка вверх вместе с курсором;

`Ctrl-R` или `PgUp` — на страницу вверх;

`Ctrl-C` или `PgDn` — на страницу вниз;

`Ctrl-Q S` или `HOME` — в начало строки;

`Ctrl-Q D` или `END` — в конец строки;

`Ctrl-Q E` или `Ctrl-HOME` — в начало экрана;

Ctrl-Q X или *Ctrl-END* — в конец экрана;
Ctrl-Q R или *Ctrl-PgUp* — в начало файла;
Ctrl-Q C или *Ctrl-PgDn* — в конец файла;
Ctrl-Q B — в начало блока;
Ctrl-Q K — в конец блока;
Ctrl-Q P — на последнюю позицию (используется после поиска или поиска/замены);
Ctrl-Q W — на последнюю ошибку.

П1.4.2. Команды удаления/вставки

Ctrl-V или *INS* — включить/отключить режим вставки;
Ctrl-N — вставить строку;
Ctrl-Y — удалить строку;
Ctrl-H или *Backspace* — стереть символ слева от курсора;
Ctrl-G или *DEL* — стереть символ над курсором;
Ctrl-T — стереть слово справа от курсора;
Ctrl-Q Y — стереть остаток строки справа от курсора.

П1.4.3. Команды работы с блоками

При подготовке текстов программ часто возникает необходимость перенести фрагмент текста в другое место или удалить его. Для такого рода операций удобно использовать блоки — фрагменты текста, рассматриваемые как единое целое. Длина блока может быть достаточно большой (до 64 Кбайт), он может занимать несколько экранных страниц. В каждый момент в среде может быть объявлен только один блок в одном окне редактора. Обмен блоками между окнами возможен только через буфер редактора (см. опцию *EDIT* в главном меню).

Ctrl-K B — пометить начало блока;
Ctrl-K K — пометить конец блока;
Ctrl-K T — пометить в качестве блока слово слева от курсора;
Ctrl-K P — напечатать блок;
Ctrl-K C — копировать блок, начиная с позиции курсора;
Ctrl-K V — переместить блок;
Ctrl-K H — убрать выделение блока цветом; повторное использование *Ctrl-K H* вновь выделит блок;
Ctrl-K Y — удалить блок;
Ctrl-K R — читать блок из дискового файла;
Ctrl-K W — записать блок на диск;
Ctrl-K I — сместить блок вправо;
Ctrl-K U — сместить блок влево.

П1.4.4. Прочие команды

Ctrl-Q F — искать по образцу;
Ctrl-L — продолжить поиск;
Ctrl-Q A — искать по образцу и заменять;
Ctrl-U — прекратить дальнейшее выполнение поиска или поиска и замены;
Ctrl-K n — установить маркер; $n = 0..9$ (см. ниже);
Ctrl-Q n — искать маркер;
Ctrl-Q W — искать ошибку;
Ctrl-Q [— искать правую парную скобку (см. ниже);
Ctrl-Q] — искать левую парную скобку (см. ниже);
Ctrl-Q L — восстановить испорченную строку (см. ниже);
Ctrl-Q T или *Ctrl-O T* — включить/отключить табуляцию;
Ctrl-O F — переключить заполнение табуляции;
Ctrl-O I или *Ctrl-Q I* — включить/отключить автоотступ;
Ctrl-O O — вставить настройку компилятора в начало файла (см. ниже).

Назначение большинства команд понятно без каких-либо комментариев, но некоторые нуждаются в пояснении.

Ctrl-K n. Устанавливает в текущую позицию курсора маркер с номером $n = 0..9$. Маркер на экране невидим и никак не влияет на исполнение программы. Команда используется со —

вместно с командой *Ctrl-Q n* (искать маркер с номером *n*) для ускорения поиска нужных фрагментов текста при разработке крупных программ. Обратите внимание — комбинации клавиш *Ctrl-K n* набираются следующим образом: вначале нажимается *Ctrl*, затем, не отпуская ее, — *K*; после этого все клавиши отпускаются и нажимается клавиша с цифрой *n*. Точно так же набирается команда *Ctrl-Q n*.

Ctrl-Q] и *Ctrl-Q [*. Эти команды используются для поиска ближайшей парной скобки. Команды позволяют отыскивать пары скобок (и), { и }, [и]. Подведите курсор так, чтобы он указывал на одну из скобок, и дайте соответствующую команду — редактор отыщет нужную парную скобку.

Ctrl-O O. Эта команда заставит редактор поместить в самое начало файла строки, со- держащие текущую настройку среды в виде директив компилятора, например:

```
{ $A+, B-, D+, E+, F+, G+, I+, L+, N+, O-, R+, S+, V+, X+ }
{ $M 16384, 0, 655360 }
{ $DEFINE single }
```

П1.4.5. Команды, передаваемые среде из редактора

Некоторые наиболее часто используемые команды можно передать среде непосредственно из режима редактирования. Все они уже описаны выше, когда рассматривалась система меню. Тем не менее я повторю их здесь, чтобы Вам было легче ими пользоваться.

F1 — получить справку;

F2 — записать файл из окна редактора на диск;

F3 — прочитать файл с диска в окно редактора;

F4 — исполнить до курсора (выполнить опцию *RUN/GO TO CURSOR*);

F5 — распахнуть окно на весь экран или вернуть ему прежние размеры;

F6 — активизировать следующее окно;

F7 — проследить процедуру (выполнить опцию *RUN/TRACE INTO*);

F8 — пропустить процедуру (выполнить опцию *RUN/STEP OVER*);

F9 — компилировать программу (выполнить опцию *COMPILE/MAKE*);

F10 — перейти в главное меню;

Ctrl-F1 — получить контекстную справку;

Ctrl-F2 — сбросить режим отладки;

Ctrl-F3 — активизировать окно программного стека;

Ctrl-F4 — вычислить выражение или показать/изменить переменную (выполнить опцию *DEBUG/EVALUATE/MODIFY*);

Ctrl-F5 — перейти к режиму установки положения и размеров окна;

Ctrl-F7 — добавить выражение в окно отладки (выполнить опцию *DEBUG/WATCHES/ ADD WATCH*);

Ctrl-F8 — переключить контрольную точку;

Ctrl-F9 — выполнить компиляцию и прогон программы;

Ctrl-Del — очистить буфер редактора;

Ctrl-Ins — копировать блок в буфер редактора;

Alt-C — вызвать меню компиляции *COMPILE*;

Alt-D — вызвать меню отладки *DEBUG*;

Alt-E — вызвать меню редактора *EDIT*;

Alt-F — вызвать меню файловой службы *FILE*;

Alt-H — вызвать меню справочной службы *HELP*;

Alt-O — вызвать меню установок *OPTIONS*;

Alt-R — вызвать меню прогона программы *RUN*;

Alt-S — вызвать меню службы поиска *SEARCH*;

Alt-W — вызвать меню окон *WINDOW*;

Alt-X — выйти из Турбо Паскаля;

Alt-0 — получить список открытых окон;

Alt-F1 — получить последнюю справку;

Alt-F3 — закрыть активное окно;

Alt-F5 — показать окно программы;

Alt-F9 — выполнить опцию *COMPILE/COMPILE*;

Shift-F1 — получить список ссылок справочной службы;

Shift-F6 — активизировать предыдущее активное окно;

Shift-Del — перенести блок из окна редактора в буфер;

Shift-Ins — копировать буфер в окно редактора.

П1.5. ВЫЗОВ ТУРБО-ПАСКАЛЯ

Полный формат команды ДОС, осуществляющей вызов Турбо Паскаля, имеет вид:

[PATH]turbo [Options] [FileName]

Здесь *PATH* — путь к системному каталогу (см. гл.1 и п.5.1.1);

Optoins — список опций;

FileName — имя *PAS*-файла.

В квадратных скобках указаны необязательные параметры команды.

Параметр *FileName*, если он указан, задает имя текстового файла, который будет автоматически загружен средой в активное окно редактора. Если в имени файла не указано расширение, среда добавляет стандартное расширение *PAS*.

С помощью параметров *Options* среде можно передать указание изменить нужным образом свою настройку. Каждый параметр задается в формате */Z*, где *Z* — буква, определяющая настройку среды (см. ниже), — знаки «+» (включить настройку) или «-» (отключить настройку); несколько параметров отделяются друг от друга пробелами.

В списке *Options* можно использовать следующие управляющие параметры:

/C<имя> — загрузить нужный файл конфигурации (эквивалентно опции *OPTIONS/OPEN* среды); *<имя>* — имя конфигурационного файла, например: */CMY.TP*;

/D+ — использовать дополнительный монитор (эквивалентно опции *OPTIONS/ENVIRONMENT/STARTUP/DUAL MONITOR SUPPORT*);

/E<размер> — установить нужный размер памяти для экранного буфера (эквивалентно опции *OPTIONS/ENVIRONMENT/STARTUP/EDITOR HEAP SIZE*); *<размер>* — устанавливаемый размер (килобайт); например */E15* — установить размер экранного буфера в 15 Кбайт;

/G+ — сохранять копию графического экрана (эквивалентно опции *OPTIONS/ENVIRONMENT/STARTUP/GRAPHICS SCREEN SAVE*);

/L+ — ПК оснащен жидкокристаллическим дисплеем (эквивалентно опции *OPTIONS/ENVIRONMENT/STARTUP/LCD COLOR SET*);

/N+ — ПК оснащен адаптером типа CGA (эквивалентно опции *OPTIONS/ENVIRONMENT/STARTUP/CGA SNOW CHECKING*);

/O<размер> — установить нужный размер памяти для хранения оверлейных модулей системы Турбо Паскаль (эквивалентно опции *OPTIONS/ENVIRONMENT/STARTUP/OVERLAY HEAP SIZE*);

/P+ — сохранять цветовую палитру экрана (эквивалентно опции *OPTIONS/ENVIRONMENT/STARTUP/EGA/VGA PALETTE SAVE*);

/S<путь> — определяет путь к «быстрому» диску (эквивалентно опции *OPTIONS/ENVIRONMENT/STARTUP/SWAP FILE DIRECTORY*);

/T+ — загружать в память библиотеку *SYSTEM.TPU* из файла *TURBO.TPL* (эквивалентно опции *OPTIONS/ENVIRONMENT/STARTUP/LOAD TURBO.TPL*);

/W<размер> — установить нужный размер памяти для хранения окон Турбо Паскаля (эквивалентно опции *OPTIONS/ENVIRONMENT/STARTUP/WINDOW HEAP SIZE*);

/X+ — использовать *EMS*-память (эквивалентно опции *OPTIONS/ENVIRONMENT/STARTUP/USE EXPANDED MEMORY*).

П2. ВАРИАНТЫ КОДИРОВКИ ЗНАКОГЕНЕРАТОРОВ ПК

Стандартный знакогенератор ПК IBM PC имеет кодировку символов, изображенную на рис.26. Символы с кодами от 0 до 127, образующие первую половину символов знакогенератора (рис.П2.1, а), построены по стандарту ASCII и одинаковы на всех IBM-совместимых ПК. Вторая половина символов (коды 128...255) могут отличаться на ПК разного типа. В стандартном знакогенераторе фирмы IBM символы псевдографики занимают три смежных колонки (коды 176...223, см. рис.П2.1, б). Колонки с кодами от 128 до 175 и от 224 до 239 используются для размещения некоторых символов национального алфавита различных европейских языков, а последняя колонка (коды 240...255) — для размещения специальных знаков. С учетом этого расположения символов разрабатывается подавляющее большинство программ зарубежного происхождения.

	000	016	032	048	064	080	096	112			128	144	160	176	192	208	224	240	
00		►		0	e	P	'	p	00		Ѓ	É	á	▤	┌	Ш	κ	≡	00
01	☐	◄	!	1	A	Q	a	q	01		ü	æ	í	▥	└	Т	β	±	01
02	⊖	‡	"	2	B	R	b	r	02		é	ff	ó	▧	┐	П	Г	≥	02
03	♥	!!	#	3	C	S	c	s	03		â	ô	ú	▨	└	Ш	π	≤	03
04	♦	¶	\$	4	D	T	d	t	04		ä	ö	ñ	▩	└	Σ	Г		04
05	♣	§	%	5	E	U	e	u	05		à	ò	ñ	▪	└	Г	σ	J	05
06	♠	▬	&	6	F	V	f	v	06		ã	û	ª	▫	└	П	μ	÷	06
07	•	±	'	7	G	W	g	w	07		ç	ù	º	▬	└	П	τ	≈	07
08	◼	↑	(	8	H	X	h	x	08		ê	ÿ	¿	▮	└	†	¤	°	08
09	◊	↓	)	9	I	Y	i	y	09		ë	ÿ	Г	▯	└	П	θ	.	09
10	◉	→	*	:	J	Z	j	z	10		è	Ü	Г	▮	└	Г	Ω	.	10
11	♂	←	+	;	K	[	k	{	11		ï	ç	½	▮	└	▮	δ	J	11
12	♀	└	,	<	L	\	l	ı	12		î	ç	¼	▮	└	▮	ω	ⁿ	12
13	ℙ	•	-	=	M	]	m	}	13		ì	¥	ı	▮	└	=	▮	²	13
14	ℙ	▲	.	>	N	^	n	~	14		ñ	R	«	▮	└	▮	€	▮	14
15	*	▼	/	?	O	_	o	△	15		ñ	f	»	└	└	▮	П		15

а)

б)

Рис.П2.1. Стандартная кодировка фирмы IBM: а) для кодов 0...127; б) для кодов 128...255

Стандартный отечественный знакогенератор строится по рекомендациям Международного консультационного комитета по телеграфии и телефонии (МККТТ). Расположение символов во второй половине таблицы этого знакогенератора (рис.П2.2,а) резко отличается от принятого фирмой IBM, что затрудняет использование зарубежного программного обеспечения на отечественных ПК. В связи с этим, стандартный (так называемый ГОСТовский) вариант кодировки часто заменяется альтернативным (рис.П2.2, б), главное достоинство которого — расположение символов псевдографики на тех же местах, что и в знакогенераторе IBM. Недостаток такого знакогенератора заключается в том, что символы кириллицы не образуют непрерывный массив. Вариант, показанный на рис.П2.2, б в настоящее время получил наибольшее распространение на отечественных ПК. Именно на него рассчитаны практически все программы отечественного производства. Он стал фактическим стандартом для зарубежных фирм, изготавливающих ПК для экспорта в нашу страну.

	128	144	160	176	192	208	224	240	
00	А	Р	а	р	Е				00
01	Б	С	б	с	ё				01
02	В	Т	в	т	у				02
03	Г	У	г	у	х				03
04	Д	Ф	д	ф	ц				04
05	Е	Х	е	х	ш				05
06	Ж	Ц	ж	ц	ч				06
07	З	Ч	з	ч	ш				07
08	И	Ш	и	ш	щ				08
09	Й	Щ	й	щ	ъ				09
10	К	Ъ	к	ъ	ь				10
11	Л	Ы	л	ы	н				11
12	М	Ь	м	ь	о				12
13	Н	Э	н	э	ю				13
14	О	Ю	о	ю	я				14
15	П	Я	п	я					15

а)

	128	144	160	176	192	208	224	240	
00	А	Р	а	р	Е				00
01	Б	С	б	с	ё				01
02	В	Т	в	т	у				02
03	Г	У	г	у	х				03
04	Д	Ф	д	ф	ц				04
05	Е	Х	е	х	ш				05
06	Ж	Ц	ж	ц	ч				06
07	З	Ч	з	ч	ш				07
08	И	Ш	и	ш	щ				08
09	Й	Щ	й	щ	ъ				09
10	К	Ъ	к	ъ	ь				10
11	Л	Ы	л	ы	н				11
12	М	Ь	м	ь	о				12
13	Н	Э	н	э	ю				13
14	О	Ю	о	ю	я				14
15	П	Я	п	я					15

б)

Рис.П2.2. Варианты кодировки для кодов 128...255: а) по рекомендациям МККТТ; б) наиболее популярный

ПЗ. СООБЩЕНИЯ И КОДЫ ОШИБОК

ПЗ.1. СООБЩЕНИЯ ОБ ОШИБКАХ ПЕРИОДА КОМПИЛЯЦИИ

Среда Турбо Паскаля дает исчерпывающую информацию о характере и месте обнаруженной компилятором ошибки. При обнаружении ошибки среда автоматически загружает в окно редактора исходный файл и помещает курсор около того места, где в исходном тексте обнаружена ошибка. При этом в верхней строке редактора появляется диагностическое сообщение. После нажатия на любую клавишу (кроме *F1*) верхняя строка восстанавливает свой первоначальный вид и среда переходит к режиму редактирования. Если после появления сообщения об ошибке нажать *F1*, на экране появится окно справочной службы с детальной информацией об ошибке и рекомендациями по ее устранению. Некоторые ошибки в исходном тексте обнаруживаются не сразу, а в ходе продолжающегося контекстного разбора. Например, несоответствие типов в операторе присваивания не может быть обнаружено до тех пор, пока не будет вычислено целиком выражение в правой части этого оператора. В таких случаях ищите ошибку слева от курсора или в предыдущей строке текста.

Ниже приводятся сообщения об ошибках в том виде, в каком они появляются в верхней строке¹ редактора, а также перевод сообщений справочной службы.

1 Out of memory (Выход за границы памяти).

Компилятору не хватает памяти. Имеется ряд возможных решений этой проблемы:

- если в опции *COMPILE/DESTINATION* установлено значение *MEMORY*, замените эту опцию на *DISK*;
- если в опции *OPTIONS/LINKER/LINK BUFFER* установлено значение *MEMORY*, замените эту опцию на *DISK*;
- если Вы используете резидентные обслуживающие программы, такие как *SIDEKICK*, *NORTON*, удалите их из памяти;
- если Вы используете интегрированную среду *TURBO.EXE*, то попробуйте воспользоваться компилятором *TPC.EXE* — он занимает меньше памяти.

Если ни одна из рекомендаций не помогает, то, возможно, Ваша программа просто слишком велика, чтобы компилировать ее в таком объеме памяти. В этом случае Вы должны разбить ее на два или более модулей.

2 Identifier expected (Не указан идентификатор).

В этом месте должен находиться идентификатор. Возможно, Вы пытаетесь использовать в качестве идентификатора зарезервированное слово.

3 Unknown identifier (Неизвестный идентификатор).

Этот идентификатор не был описан.

4 Duplicate identifier (Двойной идентификатор).

Попытка дважды описать один и тот же идентификатор.

5 Syntax error (Синтаксическая ошибка).

В исходном тексте найден недопустимый символ. Возможно, Вы забыли заключить в апострофы строковую константу.

6 Error in real constant (Ошибка в вещественной константе).

Синтаксис констант вещественного типа описан в гл.4.

7 Error in integer constant (Ошибка в целой константе).

Синтаксис констант целого типа описан в гл.4. Учтите, что после целых действительных чисел, превышающих диапазон представления целых чисел ($-2147483648..+2147483647$), должны ставиться точка и ноль, например, 12345678912.0.

¹ На экране после кода ошибки (цифры) нет точки, поэтому ее нет и в соответствующем месте текста.

8 String constant exceeds line (Строковая константа превышает допустимые размеры).

Вероятно, Вы забыли поставить апостроф в конце строковой константы.

9 Too many nested files (Слишком много вложенных файлов).

Компилятор допускает не более пяти уровней вложенности включаемых файлов (т.е. в исходном файле есть ссылка на включаемый файл, в котором есть ссылка на еще один включаемый файл и т.д.).

10 Unexpected end of file (Не найден конец файла).

Вы могли получить это сообщение об ошибке по одной из следующих причин:

- Ваш исходный файл закончился перед последним **END** основного раздела операторов; вероятно, в Вашей программе неодинаковое количество операторов **BEGIN** и **END**;
- включаемый файл заканчивается в середине раздела операторов; каждый раздел операторов должен целиком помещаться в одном файле;
- Вы не закончили комментарий.

11 Line too long (Слишком длинная строка).

Максимальная длина строки, обрабатываемой компилятором, равна 126 символам (обратите внимание: редактор среды может обрабатывать строки практически неограниченной длины).

12 Type identifier expected (Здесь нужен идентификатор типа).

Не указан тип идентификатора.

13 Too many open files (Слишком много открытых файлов).

Появление этой ошибки означает, что конфигурационный файл *CONFIG.SYS* операционной системы не включает параметр *FILES=xx* или этот параметр указывает слишком мало файлов. Увеличьте число файлов до нужного значения, например, до 80.

14 Invalid file name (Неверное имя файла).

Имя файла неверно или указан несуществующий путь.

15 File not found (Файл не найден).

Файл не был найден в просмотренных каталогах.

16 Disk full (Диск заполнен).

Удалите некоторые файлы или воспользуйтесь новым диском.

17 Invalid compiler directive (Неправильная директива компилятора).

Неверная буква в директиве компилятора, один из параметров директивы компилятора неверный, или Вы пользуетесь глобальной директивой компилятора, когда компиляция тела программы уже началась.

18 Too many files (Слишком много файлов).

В компиляции программы или программного модуля участвуют слишком много файлов. Попробуйте не использовать так много файлов, например, объединяя включаемые файлы или делая короче имена файлов.

19 Undefined type in pointer definition (Неопределенный тип в объявлении указателя).

Попытка объявить типизированный указатель, связанный с ранее не объявленным типом данных.

20 Variable identifier expected (Отсутствует идентификатор переменной).

На этом месте должен быть идентификатор переменной.

21 Error in type (Ошибка в объявлении типа).

Объявление типа не может начинаться с этого символа.

22 Structure too large (Слишком большая структура).

Максимально допустимый размер любого структурного типа составляет 65520 байт.

23 Set base type of range (Базовый тип множества нарушает границы).

Базовый тип множества должен представлять собой тип — диапазон с границами в пределах от 0 до 255 или перечисляемый тип с не более чем 256 значениями.

24 File components may not be files (Компонентами файла не могут быть файлы).

Конструкции типа **file of file** (файл файлов) или **file of object** (файл объектов) не допускаются. Нельзя объявлять любые структурные типы, которые используют в качестве компонентов объекты или файлы.

25 Invalid string length (Неверная длина строки).

Объявляемая длина строки должна находиться в диапазоне от 1 до 255.

26 Type mismatch (Несоответствие типов).

Это сообщение может быть вызвано следующими причинами:

- несовместимые типы переменной и выражения в операторе присваивания;
- несовместимые типы фактического и формального параметров в обращении к процедуре или функции;
- тип выражения не совместим с типом индекса при индексировании массива;
- несовместимые типы операндов в выражении.

27 Invalid subrange base type (Неправильный базовый тип для типа-диапазона).

Допустимыми базовыми типами являются все порядковые типы.

28 Lower bound greater than upper bound (Нижняя граница больше верхней).

Описание типа-диапазона содержит неправильные границы.

29 Ordinal type expected (Нужен порядковый тип).

Вещественные, строковые, структурные, процедурные типы и указатели в данном месте программы не допускаются.

30 Integer constant expected (Нужна целая константа).

31 Constant expected (Нужна константа).

32 Integer or real constant expected (Нужна целая или вещественная константа).

33 Type identifier expected (Нужен идентификатор типа)

34 Invalid function result type (Неправильный тип результата функции)

Правильными типами результата функции являются все простые типы, строковые типы и указатели.

35 Label identifier expected (Нужен идентификатор метки).

Метка не обозначена с помощью идентификатора, как это требуется из контекста программы.

36 BEGIN expected (Нужен BEGIN).

37 END expected (Нужен END).

38 Integer expression expected (Нужно выражение типа INTEGER).

39 Ordinal expression expected (Нужно выражение перечисляемого типа).

40 Boolean expression expected (Нужно выражение типа BOOLEAN).

41 Operand types do not match operator (Типы операндов не соответствуют операции).

Данная операция не может быть применена к указанным операндам, например, 'A' div '2'.

42 Error in expression (Ошибка в выражении).

Данный символ не может участвовать в выражении указанным образом. Возможно, Вы забыли указать операцию между двумя операндами.

43 Illegal assignment (Неверное присваивание).

Файлам и нетипизированным переменным нельзя присваивать значения. Идентификатору функции можно присвоить значение только внутри раздела операторов данной функции.

44 Field identifier expected (Нужен идентификатор поля).

Попытка использовать запись целиком в том месте, где требуется ссылка на какое-либо поле записи.

45 Object file too large (Объектный файл слишком большой).

Турбо Паскаль не может компоновать файлы .OBJ больше 64 Кбайт.

46 Undefined external (Неопределенная внешняя процедура).

Внешняя процедура или функция не имеет соответствующего определения *PUBLIC* в объектном файле. Убедитесь, что Вы указали все объектные файлы в директивах *{\$L <имя .OBJ-файла>}* и проверьте написание идентификаторов процедуры или функции в файле .ASM.

47 Invalid object file record (Неправильная запись объектного файла).

Файл .OBJ содержит неверную объектную запись. Убедитесь, что данный файл является действительно файлом .OBJ.

48 Code segment too large (Сегмент кода слишком большой).

Максимальный размер кода программы или программного модуля равняется 65520 байтам. Разбейте Вашу программу или модуль на два или более модулей.

49 Data segment too large (Сегмент данных слишком велик).

Максимальный размер сегмента данных программы равен 65520 байтам, включая данные, используемые программными модулями. Если Вам нужно большее количество глобальных данных, опишите большие структуры с помощью указателей и выделяйте для них память динамически с помощью процедуры *NEW*.

50 DO expected (Нужен оператор DO).

51 Invalid PUBLIC definition (Неверное PUBLIC-определение).

Возможные причины сообщения:

- данный идентификатор получил тип *PUBLIC* с помощью соответствующей директивы языка ассемблера, но не соответствует описанию *EXTERNAL* в программе или программном модуле Паскаля;
- две или более директивы *PUBLIC* языка ассемблера определяют один и тот же идентификатор;
- файлы .OBJ определяют символы *PUBLIC*, не находящиеся в сегменте *CODE*.

52 Invalid EXTRN definition (Неправильное EXTRN-определение).

Возможные причины сообщения:

- программа на ассемблере ссылается с помощью директивы *EXTRN* на идентификатор, который не описан в программе на Паскале и не был описан в интерфейсных секциях используемых программных модулей;
- ассемблерная программа ссылается на идентификатор, обозначающий абсолютную переменную (т.е. определенную словом *ABSOLUTE*);
- ассемблерная программа ссылается на идентификатор процедуры или функции типа *INLINE*.

53 Too many EXTRN definition (Слишком много EXTRN-определений).

Турбо Паскаль не может обрабатывать файлы .OBJ при более чем 256 определениях *EXTRN*.

54 OF expected (Требуется OF).

55 INTERFACE expected (Требуется интерфейсная секция).

56 Invalid relocatable reference (Неправильная перемещаемая ссылка).

Возможные причины сообщения:

- файл .OBJ содержит данные и перемещаемые ссылки в сегментах, отличных от *CODE*. Например, Вы пытаетесь описать инициализированные переменные в сегменте *DATA*;

- файл *.OBJ* содержит ссылки с размерами в байтах на перемещаемые символы. Такая ошибка происходит в случае, если Вы используете операторы *HIGH* и *LOW* с перемещаемыми символами или если Вы ссылаетесь в директивах *DB* на перемещаемые символы;
- операнд ссылается на перемещаемый символ, который не был определен в сегменте *CODE* или в сегменте *DATA*;
- операнд ссылается на процедуру *EXTRN* или функцию *EXTRN* со сдвигом, например, *CALL SortProc+8*.

57 THEN expected (Требуется THEN).

58 TO or DOWNT0 expected (Требуется TO или DOWNT0).

59 Undefined forward (Неопределенное опережающее описание).

Возможные причины сообщения:

- была описана процедура или функция в интерфейсной секции программного модуля, но ее определение отсутствует в секции реализации;
- процедуры или функции были описаны с помощью опережающего описания, но их определение не найдено.

60 Too many procedures (Слишком много процедур).

Турбо Паскаль допускает не более 512 процедур или функций в одном модуле. Если Вы компилируете программу, то поместите некоторые процедуры или функции в модули. Если Вы компилируете модуль, то разбейте его на два или несколько модулей.

61 Invalid typecast (Неверное преобразование типа).

Возможные причины сообщения:

- Вы пытаетесь разместить в памяти, занимаемой некоторой переменной, значение выражения другого типа в случае, когда размер размещаемого значения не равен размеру переменной;
- Вы пытаетесь осуществить преобразование типа выражения, когда разрешается только ссылка на переменную, процедуру или функцию.

62 Division by zero (Деление на ноль).

Предшествующая операция пытается выполнить деление на ноль.

63 Invalid file type (Неверный файловый тип).

Данный файловый тип не обслуживается процедурой обработки файлов. Например, процедура *READLN* используется для типизированного файла или процедура *SEEK* — для текстового файла.

64 Cannot Read or Write variables of this type (Нет возможности считать или записать переменные данного типа).

Нарушены следующие ограничения:

- процедуры *READ* и *READLN* могут считывать переменные символьного, целого, действительного и строкового типов;
- процедуры *WRITE* и *WRITELN* могут выводить переменные символьного, целого, действительного, булевского и строкового типов.

65 Pointer variable expected (Нужно использовать переменную-указатель).

Предыдущая переменная должна быть указателем.

66 String variable expected (Нужна строковая переменная).

Предшествующая переменная должна иметь строковый тип.

67 String expression expected (Нужно выражение строкового типа).

Предшествующее выражение должно иметь строковый тип.

68 Circular unit reference (Перекрестная ссылка модулей).

Два модуля ссылаются друг на друга:

```
Unit A;          Unit B;
Uses B;          Uses A;
.....          .....
```

69 Unit name mismatch (Несоответствие имен программных модулей).

Имя программного модуля, найденное в файле .TPU, не соответствует имени, указанному в предложении **USES**.

70 Unit version mismatch (Несоответствие версий модулей).

Один или несколько программных модулей, используемых данной программой, были изменены после их компиляции. Воспользуйтесь опцией **COMPILE/MAKE** или **COMPILE/BUILD** в интегрированной среде или опциями **/M** или **/B** в компиляторе **TPC**, что позволит автоматически скомпилировать программные модули, нуждающиеся в перекомпиляции.

71 Duplicate unit name (Повторное имя программного модуля).

Вы уже указали этот программный модуль в операторе **USES**.

72 Unit file format error (Ошибка формата файла модуля).

Файл .TPU не соответствует стандарту Турбо Паскаля.

73 IMPLEMENTATION expected (Отсутствует исполняемая часть модуля).

74 Constant and case types do not match (Типы констант и тип выражения оператора **CASE** не соответствуют друг другу).

Тип константы оператора **CASE** не совместим с выражением в операторе варианта.

75 Record variable expected (Нужна переменная типа запись).

Предшествующая переменная должна иметь тип запись.

76 Constant out of range (Константа нарушает границы).

Возможные причины сообщения:

- Вы пытаетесь указать индекс массива, выходящий за его границы;
- Вы пытаетесь присвоить переменной значение, выходящее за границы, допустимые для типа этой переменной;
- Вы пытаетесь передать в качестве фактического параметра процедуре или функции константу, выходящую за границы, допустимые для типа соответствующего формального параметра.

77 File variable expected (Нужна файловая переменная).

Предшествующая переменная должна иметь файловый тип.

78 Pointer expression expected (Нужно выражение типа указатель).

Предшествующее выражение должно иметь тип указателя.

79 Integer or real expression expected (Нужно выражение вещественного или целого типа).

Предшествующее выражение должно иметь тип **REAL** или **INTEGER**.

80 Label not within current block (Метка не находится внутри текущего блока).

Оператор **GOTO** не может ссылаться на метку, находящуюся вне текущего блока.

81 Label already defined (Метка уже определена).

Данная метка уже помечает оператор.

82 Undefined label in processing statement part (Неопределенная метка в предшествующем разделе операторов).

Данная метка была описана, и на нее осуществлялась ссылка в предшествующем разделе операторов, но она не указана в тексте программы.

83 Invalid @ argument (Неправильный аргумент операции @).

Правильными аргументами являются идентификаторы переменных, процедур и функций.

84 Unit expected (Нужно кодовое слово **UNIT**).

85«;» expected (Нужно указать«;»).

86«:» expected (Нужно указать«:»).

87«,» expected (Нужно указать«,»).

88 «(» expected (Нужно указать «(»).

89 «)» expected (Нужно указать «)»).

90 «=» expected (Нужно указать «=»).

91 «:=» expected (Нужно указать «:=»).

92 «[» or «(.» expected (Нужно указать «[» или «(.»).

93 «]» or «.)» expected (Нужно указать «]» или «.)»).

94 «.» expected (Нужно указать «.»).

95 «..» expected (Нужно указать «..»).

96 Too many variables (Слишком много переменных).

Нарушены следующие ограничения:

- общий размер глобальных переменных, описанных в программе или программном модуле, не может превышать 64 Кбайт;
- размер локальных переменных, описанных в процедуре или функции, не может превышать 64 Кбайт.

97 Invalid FOR control variable (Неправильный параметр цикла оператора FOR).

Параметр цикла оператора **FOR** должен быть переменной порядкового типа, определенной в разделе описаний текущей подпрограммы.

98 Integer variable expected (Нужна переменная целого типа).

Предшествующая переменная должна иметь целый тип.

99 File and procedure types are not allowed here (Здесь не могут использоваться файлы или процедурные типы).

Типизированная константа не может иметь файловый или процедурный тип.

100 String length mismatch (Несоответствие длины строки).

Длина строковой константы не соответствует количеству элементов символьного массива.

101 Invalid ordering of fields (Неверный порядок полей).

Поля в константе типа запись должны записываться в порядке их описания.

102 String constant expected (Нужна константа строкового типа).

103 Integer or real variable expected (Нужна переменная типа INTEGER или REAL).

Предшествующая переменная должна иметь целый или вещественный тип.

104 Ordinal variable expected (Нужна переменная порядкового типа).

Предшествующая переменная должна иметь порядковый тип.

105 INLINE error (Ошибка в операторе INLINE).

Оператор «<» не допускается в сочетании с перемещаемыми ссылками на переменные. Такие ссылки всегда имеют размер в слово.

106 Character expression expected (Предшествующее выражение должно иметь символьный тип).

107 Too many relocation items (Слишком много перемещаемых элементов).

Размер таблицы перемещения файла **.EXE** превышает 64 Кбайта, что является верхним пределом в Турбо Паскале. Если Вы обнаружили эту ошибку, то это значит, что программа просто слишком велика для обработки компоновщиком Турбо Паскаля. Возможно также, что она слишком велика для выполнения в MS-DOS. В таком случае нужно выделить в программе основной раздел, который выполнял бы обращение к двум или более вспомогательным разделам с помощью процедуры **EXEC** из модуля DOS (см. гл.11).

108 Overflow in arithmetic operator (Переполнение при выполнении арифметического оператора).

Результат предыдущей арифметической операции не лежит в диапазоне $-2\ 146\ 483\ 648 \dots +2\ 147\ 483\ 647$. Исправьте операцию или используйте вещественные типы вместо целочисленных.

109 No enclosing FOR, WHILE or REPEAT statment (Нет операторов, заканчивающих операторы FOR, WHILE или REPEAT).

Процедуры BREAK и CONTINUE не могут вызываться вне тела оператора цикла.

110 Debug information table overflow (Переполнение информационной таблицы отладки).

Возможно, программа содержит более 65536 имен или 65536 строк. Отключите генерацию таблиц отладки директивой компилятора `{SD-}` или исправьте один или более модулей.

111

Ошибка с этим кодом не описана в версии 7.0 системы Турбо Паскаль.

112 CASE constant out of range (Константа CASE нарушает допустимые границы).

Целочисленные константы оператора CASE должны находиться в диапазоне от -32768 до 32767 .

113 Error in statement (Ошибка в операторе).

Данный символ не может быть первым символом в операторе.

114 Cannot call an interrupt procedure (Невозможно вызвать процедуру прерывания).

Вы не можете непосредственно вызвать процедуру обработки прерывания.

115

Ошибка с этим кодом не описана в версии 7.0 системы Турбо Паскаль.

116 Must be in 8087 mode to compile this (Для компиляции необходим режим 8087).

Данная программа может быть скомпилирована только в режиме `{SN+}`. В состоянии `{SN-}` операции с типами SINGLE, DOUBLE, EXTENDED и COMP не допускаются.

117 Target address not found (Указанный адрес не найден).

Команда COMPILE/FIND ERROR в среде Турбо Паскаля (или поиск с помощью опции /F в командной строке компилятора TPC.EXE) не обнаружила оператор, соответствующий заданному адресу.

118 Include files are not allowed here (Здесь не допускаются включаемые файлы).

Раздел операторов должен целиком размещаться в одном файле.

119 No inherited methods are accessible here (В этом месте программы нет унаследованных методов).

Вы используете зарезервированное слово **INHERITED** вне метода объекта или в методе, который не унаследован от родительского объекта.

120

Ошибка с этим кодом не описана в версии 7.0 системы Турбо Паскаль.

121 Invalid qualifier (Неверный квалификатор).

Возможные причины сообщения:

- Вы пытаетесь индексировать переменную, которая не является массивом;
- Вы пытаетесь указать поля в переменной, которая не является записью;
- Вы используете в качестве указателя переменную, которая не является указателем.

122 Invalid variable reference (Недействительная ссылка на переменную).

Предыдущая конструкция удовлетворяет синтаксису ссылки на переменную, но она не указывает адрес памяти. Возможно Вы вызываете функцию — указатель, но забываете сослаться на результат с помощью знака `^`.

123 Too many symbols (Слишком много символов).

Программа или программный модуль содержат более 64 Кбайт символов. Если Вы компилируете программу с директивой `{SD+}`, то попробуйте отключить эту директиву или разбейте программу на несколько модулей.

124 Statement part too large (Слишком большой раздел операторов).

Турбо Паскаль ограничивает размер раздела операторов до величины примерно 24 Кбайта. Если Вы обнаружили эту ошибку, поместите части раздела операторов в одну или несколько процедур и вообще сделайте Вашу программу более структурированной.

125

Ошибка с этим кодом не описана в версии 7.0 системы Турбо Паскаль.

126 Files must be var parameters (Файлы должны передаваться как параметры-переменные).

Вы пытаетесь передать процедуре или функции параметр — значение файлового типа. Параметры файлового типа должны быть параметрами — переменными.

127 Too many conditional symbols (Слишком много условных символов).

Недостаточно памяти для определения условных символов (слов, управляющих командами условной компиляции). Попытайтесь удалить некоторые символы или уменьшить их длину.

128 Misplaced conditional directive (Пропущена условная директива).

Компилятор обнаружил директиву `{ELSE}` или `{ENDIF}` без соответствующих директив `{IFDEF}`, `{IFNDEF}` или `{IFOPT}`.

129 ENDIF directive missing (Пропущена директива ENDIF).

Исходный файл закончился внутри конструкции условной компиляции. В исходном файле должно быть равное количество директив `{IFxxx}` и `{ENDIF}`.

130 Error in initial conditional defines (Ошибка в условных определениях).

Исходные условные символы, указанные в опции `OPTIONS/COMPILER /CONDITIONAL DEFINES` являются недействительными. Турбо Паскаль требует нуля или более идентификаторов, разделенных пробелами, запятыми или точками с запятой.

131 Header does not match previous definition (Заголовок не соответствует предыдущему определению).

Возможные причины сообщения:

- заголовок процедуры или функции, указанный в интерфейсной секции, не соответствует заголовку в исполняемой части.
- заголовок процедуры или функции, указанный с помощью опережающего описания `FORWARD`, не соответствует заголовку найденной далее одноименной процедуры или функции.

132 Critical disk error (Критическая ошибка диска).

Во время компиляции произошла критическая ошибка диска (например, дисковод находится в состоянии «не готов»).

133 Cannot evaluate this expression (Нельзя вычислить данное выражение).

В выражении — константе или в отладочном выражении Вы пытаетесь использовать не поддерживаемые средства, например, в описании константы пытаетесь использовать функцию `SIN` или вызвать в отладочном выражении определенную пользователем функцию.

134 Expression incorrectly terminated (Некорректное завершение выражения).

Контекстуально в данном месте программы должен быть конец выражения или оператора.

135 Invalid format specifier (Неверный спецификатор формата).

Используется неверный спецификатор формата или числовой аргумент спецификатора формата выходит за допустимые границы.

136 Invalid indirect reference (Недопустимая косвенная ссылка).

Оператор пытается осуществить недопустимую косвенную ссылку. Например, Вы используете абсолютную переменную, базовая переменная которой в текущем модуле неизвестна, или используете программу типа *INLINE*, в которой делается ссылка на переменную, неопределенную в текущем модуле.

137 Structured variable are **not** allowed here (Здесь нельзя использовать переменную структурного типа).

Делается попытка выполнить над переменной структурного типа неподдерживаемую операцию. Например, Вы пытаетесь перемножить две записи.

138 Cannot evaluate without System **unit** (Нельзя вычислить выражение без модуля *SYSTEM*).

Чтобы отладчик смог вычислить выражение, в файле *TURBO.TPL* должен содержаться модуль *SYSTEM*.

139 Cannot access this symbol (Нет доступа к данному символу).

Как только Вы скомпилируете программу, все множество ее символов становится доступным. Однако к отдельным символам (например, к переменным) нельзя получить доступ, пока Вы не запустите программу.

140 Invalid floating-point operation (Недопустимая операция с плавающей запятой).

При выполнении операции с плавающей запятой произошло переполнение или деление на ноль.

141 Cannot compile overlay to memory (Нельзя выполнить компиляцию оверлейных модулей в память).

Программа, использующая оверлейные модули, должна компилироваться на диск.

142 Procedure or function variable expected (Должна использоваться переменная процедурного типа).

В этом контексте оператор получения адреса @ может использоваться только с переменной процедурного типа.

143 Invalid procedure or function reference (Недопустимая ссылка на процедуру или функцию).

Возможные причины сообщения:

- Вы пытаетесь вызвать процедуру в выражении;
- процедура или функция, использующаяся в качестве параметра вызова другой процедуры или функции, должна компилироваться в состоянии *{F+}* и не может описываться с помощью зарезервированных слов *INLINE* или *INTERRUPT*.

144 Cannot overlay this unit (Этот модуль не может использоваться в качестве оверлейного).

Попытка использовать в качестве оверлейного модуль, который не был скомпилирован с директивой *{O+}*.

145 Too many nested scopes (Слишком много вложений).

В программе не может быть больше 512 вложений с не более чем 128 вложениями в каждом модуле. Вложениями считаются:

- каждый модуль в предложении *USES*;
- каждая вложенная запись в типе *RECORD*;
- каждый вложенный оператор *WITH*.

146 File access denied (Отказано в доступе к файлу).

Возможные причины:

- Вы пытаетесь использовать файл с атрибутом «только для чтения» в качестве выводного файла;
- Вы используете имя каталога вместо имени выводного файла.

147 Object type expected (Здесь должен быть тип *OBJECT*).

Этот идентификатор должен принадлежать к типу *OBJECT*.

148 Local object types are not allowed (Нельзя объявлять локальные объекты).

Нельзя объявить объект в процедуре (функции).

149 VIRTUAL expected (Пропущено слово VIRTUAL).

150 Method identifier expected (Пропущен идентификатор инкапсулированного правила).

151 Virtual constructor are not allowed (Конструктор не может быть виртуальным).

153 Destructor identifier expected (Пропущен идентификатор деструктора).

154 Fail only allowed within constructor (Обращение к стандартной процедуре FAIL может содержаться только в конструкторе).

155 Invalid combination of opcode and operands (Недопустимая комбинация кода команды и операндов).

Код ассемблерной команды не может иметь такие операнды. Причины ошибки:

- указано слишком много или слишком мало операндов для данной команды, например, INC AX, BX или MOV AX;
- количество операндов правильное, но их типы или порядок следования не соответствуют данной команде, например, DEC 1, MOV AX, CL или MOV 1, AX.

156 Memory reference expected (Отсутствует адрес).

Операнд ассемблерной команды не содержит адрес. Вероятно, Вы забыли выделить квадратными скобками индексный регистр, например, MOV AX, BX+SI вместо MOV AX, [BX+SI].

157 Cannot add or subtract relocatable symbols (Нельзя складывать или вычитать перемещаемые символы).

В ассемблерных выражениях обоими операндами в операциях сложения или вычитания могут быть только ссылки на константу. Идентификаторы переменных, процедур, функций и меток являются перемещаемыми символами и не могут использоваться в качестве двух операндов одновременно в таких операциях. Если VAR — идентификатор переменной, а CONST — константы, то предложения MOV AX, CONST+CONST и MOVE AX, VAR+CONST будут правильными, в то время как выражение MOV AX, VAR+VAR недопустимо.

158 Invalid register combination (Недопустимая комбинация регистров).

Допустимыми комбинациями индексных регистров являются [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI] и [BP+DI]. Другие комбинации (например, [AX], [BP+BX], [SI+DX]) недопустимы. Заметим, что локальные переменные размещаются в стеке и доступ к ним организуется через регистр BP. Ассемблер автоматически добавляет [BP] в ссылки на такие переменные, поэтому в операндах типа LOCAL[BX], где LOCAL — локальная переменная, образуется недопустимая ссылка LOCAL[BP+BX].

159 286/287 instructions are not enabled (Недоступен набор команд микропроцессоров 286/287).

Используйте директиву {\$G+}, но учтите, что такую программу можно исполнять только на ПК, оснащенных микропроцессором Intel 80286/80287 или более поздним.

160 Invalid symbol reference (Недопустимая ссылка на символ).

К указанному символу нельзя обратиться в ассемблерной программе. Это может быть следствием таких причин:

- Вы пытаетесь использовать ссылки на стандартные процедуры (функции) или специальные массивы MEM, MEMW, MEML, PORT и PORTW в ассемблерных операндах;
- Вы обращаетесь к строковой, вещественной константе или константе типа SET в операнде ассемблерной команды;
- в ассемблерном операнде Вы обращаетесь к процедуре или функции, написанной в режиме INLINE;
- Вы пытаетесь получить с помощью операции @Result доступ к результату, возвращаемому функцией;

- Вы пытаетесь использовать короткую форму команды *JMP* для передачи управления в немеченный оператор программы.

161 Code generation error (Ошибка генерации кода).

Ошибка возникает, в частности, при компиляции ассемблерных фрагментов, содержащих команды *LOOP*, *LOOPE*, *LOOPNE*, *JCXZ*, если команда ссылается на недоступную метку.

162 ASM expected (Отсутствует зарезервированное слово *ASM*).

ПЗ.2. ОШИБКИ, ВОЗНИКАЮЩИЕ ВО ВРЕМЯ ВЫПОЛНЕНИЯ ПРОГРАММ

Некоторые ошибки, обнаруженные во время выполнения программы, приводят к появлению на экране сообщения вида

```
Runtime error nnn at xxxx:yyyy
(Ошибка периода исполнения nnn по адресу xxxx:yyyy)
```

где *nnn* — номер ошибки; *xxxx:yyyy* — адрес (сегмент и смещение). После этого сообщения программа завершает свою работу.

Ошибки периода исполнения делятся на четыре категории: ошибки, обнаруживаемые ДОС (коды ошибок с 1 до 99), ошибки ввода-вывода (с 100 по 149), критические ошибки (с 150 по 199) и фатальные ошибки (коды ошибок с 200 до 255).

ПЗ.2.1. Ошибки, обнаруживаемые ДОС

1 Invalid function number (Неверный номер функции).

Вы пытаетесь вызвать несуществующую функцию ДОС.

2 File not found (Не найден файл).

Ошибка генерируется процедурами *RESET*, *APPEND*, *RENAME* или *ERASE* в случае, если имя, присвоенное файловой переменной, указывает несуществующий файл.

3 Path not found (Путь не найден).

Ошибка генерируется процедурами:

- *RESET*, *REWRITE*, *APPEND* или *ERASE* в случае, если имя, присвоенное файловой переменной, является недействительным или указывает на несуществующий подкаталог;
- *CHDIR*, *MKDIR* или *RMDIR* в случае, если путь является недействительным или указывает несуществующий подкаталог.

4 Too many open files (Слишком много открытых файлов).

Ошибка генерируется процедурами *RESET*, *REWRITE* или *APPEND* в случае, если программа имеет слишком много открытых файлов. ДОС не позволяет использовать более 15 открытых файлов для каждого процесса. Если ошибка возникла при наличии менее 15 открытых файлов, то она может указывать на то, что файл *CONFIG.SYS* не содержит параметра *FILES=xxx* или что этот параметр задает слишком мало файлов. Увеличьте параметр *FILES=xxx* до какого-либо подходящего значения, например, до 80.

5 File access defined (Отказано в доступе к файлу).

Данная ошибка генерируется процедурами:

- *RESET* или *APPEND* в случае, когда имя, присвоенное файловой переменной, указывает каталог или файл, доступный только для чтения, в то время как параметр *FILEMODE* файловой переменной содержит указание на запись данных;
- *REWRITE* в случае, когда каталог заполнен, или если имя, присвоенное файловой переменной, задает каталог или существующий файл, доступный только для чтения;
- *RENAME* в случае, если имя, присвоенное файловой переменной, указывает каталог или если новое имя указывает существующий файл;
- *ERASE* в случае, если имя, присвоенное файловой переменной, указывает каталог или файл, доступный только для чтения;

- **MKDIR** в случае, если файл с тем же именем уже существует в порождающем каталоге, если нет места в порождающем каталоге или если путь к каталогу содержит имя логического устройства;
- **RMDIR** в случае, если каталог не является пустым, если путь не определяет каталог или если путь задает корневой каталог;
- **READ** или **BLOCKREAD** в случае типизированного или нетипизированного файла, если файл не открыт для чтения;
- **WRITE** или **BLOCKWRITE** для типизированного или нетипизированного файла в случае, если этот файл не открыт для записи.

6 Invalid **file** handle (Недопустимый файловый канал).

Данная ошибка генерируется в случае, когда системному вызову **DOC** передается недопустимый файловый канал. Эта ошибка не должна возникать в правильно работающей программе. Ее появление является свидетельством того, что файловая переменная каким-либо образом испорчена.

12 Invalid file access code (Недействительный код доступа к файлам).

Ошибка генерируется процедурами **RESET** или **APPEND**, если значение параметра **FILEMODE** в файловой переменной не является допустимым.

15 Invalid drive number (Недопустимый номер дисководов).

Ошибка генерируется процедурой **GETDIR** в случае, если номер дисковода не является допустимым.

16 Cannot remove current directory (Нельзя удалить текущий каталог).

Ошибка генерируется процедурой **RMDIR** в случае, если путь указывает текущий каталог.

17 Cannot rename across drives (Нельзя при переименовании указывать разные дисководы).

Генерируется процедурой **RENAME** в случае, если оба файла не находятся на одном и том же диске.

П3.2.2. Ошибки ввода-вывода

Если один из операторов компилировался с директивой **{SI+}**, то ошибка ввода-вывода приводит к прекращению выполнения программы. В состоянии **{SI-}** программа продолжает выполняться, а ошибка возвращается функцией **IORESULT**.

100 Disk read error (Ошибка чтения с диска).

Генерируется процедурой **READ** в типизированном файле в случае, если Вы пытаетесь осуществить считывание из полностью прочитанного файла.

101 Disk write error (Ошибка записи на диск).

Ошибка генерируется процедурами **CLOSE**, **WRITE**, **WRITELN**, **FLUSH** в случае, если диск заполнен.

102 File not assigned (Файлу не присвоено имя).

Ошибка генерируется процедурами **RESET**, **REWRITE**, **APPEND**, **RENAME** и **ERASE** в случае, если файловой переменной не было присвоено имя файла с помощью обращения к процедуре **ASSIGN**.

103 File not open (Файл не открыт).

Ошибка генерируется процедурами **CLOSE**, **READ**, **WRITE**, **SEEK**, **EOF**, **FILEPOS**, **FILESIZE**, **FLUSH**, **BLOCKREAD**, **BLOCKWRITE** в случае, если файл не открыт.

104 File not open for input (Файл не открыт для ввода).

Ошибка генерируется процедурами **READ**, **READLN**, **EOF**, **EOLN**, **SEEKEOF** или **SEEKEOLN** в текстовом файле в случае, если файл не открыт для ввода.

105 File not open for output (Файл не открыт для вывода).

Ошибка генерируется процедурами **WRITE** или **WRITELN** в текстовом файле в случае, если файл не открыт для вывода.

106 Invalid numeric format (Неверный числовой формат).

Генерируется процедурами *READ* или *READLN* в случае, если числовое значение, считанное из текстового файла, не соответствует правильному числовому формату.

ПЗ.2.3. Критические ошибки

150 Disk is write protected (Диск защищен от записи).

151 Unknown unit (Неизвестный модуль).

152 Drive not ready (Дискковод находится в состоянии «не готов»).

153 Unknown command (Неопознанная команда).

154 CRC error in data (Ошибка в исходных данных).

155 Bad drive request structure length (При обращении к диску указана неверная длина структуры).

156 Disk seek error (Ошибка при операции установки головок на диске).

157 Unknown media type (Неизвестный тип носителя).

158 Sector not found (Сектор не найден).

159 Printer out of paper (Кончилась бумага на принтере).

160 Device write fault (Ошибка при записи на устройство).

161 Device read fault (Ошибка при чтении с устройства).

162 Hardware failure (Сбой аппаратуры).

ПЗ.2.4. Фатальные ошибки

Эти ошибки всегда приводят к немедленной остановке программы.

200 Division by zero (Деление на ноль).

201 Range check error (Ошибка при проверке границ).

Ошибка генерируется операторами, скомпилированными в состоянии *{SR+}*, при возникновении одной из следующих ситуаций:

- индексное выражение массива находилось вне допустимого диапазона;
- была осуществлена попытка присвоить переменной значение, находящееся вне диапазона переменной;
- была осуществлена попытка передать значение, находящееся вне допустимого диапазона, в качестве параметра процедуре или функции.

202 Stack overflow error (Переполнение стека).

Эта ошибка генерируется на входе в процедуру или функцию, скомпилированную в режиме *{S+}*, в случае, если нет достаточной области для размещения локальных переменных подпрограммы. Увеличьте размер стека, используя директиву компилятора *{M}*.

203 Heap overflow error (Переполнение кучи).

Эта ошибка генерируется процедурами *NEW* или *GETMEM* в случае, если в куче нет свободной памяти требуемого размера.

204 Invalid pointer operation (Недействительная операция с указателем).

Эта ошибка генерируется процедурами *DISPOSE* или *FREEMEM* в случае, когда указатель имеет значение *NIL* или указывает на адрес, лежащий за пределами динамически распределяемой области памяти.

205 Floating point overflow (Переполнение при операции с плавающей запятой).

В результате выполнения операции с плавающей запятой получено слишком большое вещественное число.

206 Floating point underflow (Исчезновение порядка при операции с плавающей запятой).

Эта ошибка генерируется только в том случае, если используется сопроцессор 8087/80287/80387 с управляющим словом, которое демаскирует ошибку исчезновения порядка. По умолчанию исчезновение порядка приводит к возвращению результата, равного нулю.

207 Invalid floating point operation (Недопустимая операция с плавающей запятой).

Возможные причины сообщения:

- аргумент функций *TRUNC* или *ROUND* не может быть преобразован в целое число, находящееся внутри диапазона типа *LONGINT* (от -2147483648 до +2147483647);
- отрицательный аргумент функции *SQRT* (извлечение квадратного корня);
- аргумент функции *LN* (логарифм) равен нулю или имеет отрицательное значение;
- произошло переполнение стека сопроцессора.

208 Overlay manager **not** installed (Не установлена подсистема управления оверлеем).

Ваша программа вызывает оверлейную процедуру или функцию, а подсистема управления оверлеем не инициализирована. Вероятнее всего, в программе отсутствует обращение к процедуре *OVRINIT* или обращение к этой процедуре завершилось с ошибкой. Нужно иметь в виду, что если в каком-либо из оверлейных модулей содержится раздел инициации, то в программе необходимо создать дополнительный или использовать имеющийся неоверлейный модуль, вызывающий процедуру *OVRINIT* в своем разделе инициализации, и указать этот модуль в предложении *USES* перед любым из оверлейных модулей.

209 Overlay file read error (Ошибка чтения оверлейного файла).

Ошибка чтения произошла, когда подсистема управления оверлеем пыталась считать оверлейный модуль из оверлейного файла.

210 Object not initialized (Не инициализирован объект).

Вы обращаетесь к виртуальному правилу применительно к неинициализированному объекту (до вызова конструктора).

211 Call to abstract method (Вызов абстрактного правила).

Эта ошибка генерируется правилом *ABSTRACT* модуля *OBJECT* при работе в среде Turbo Vision в случае обращения к абстрактному правилу, т.е. к виртуальному правилу, которое разработано специально для его замены в объектах — потомках.

212 Stream registration error (Ошибка в регистрируемом потоке).

Эта ошибка генерируется правилом *REGISTERTYPE* модуля *OBJECT* при работе в среде Turbo Vision в случае возникновения одной из следующих ошибок:

- запись регистрируемого потока не содержит сегмента данных;
- поле *OBJTYPE* записи регистрируемого потока нулевое;
- указанный тип уже был зарегистрирован;
- существует другой тип с таким же полем *OBJTYPE*.

213 Collection index out of range (Набираемый индекс выходит из границ диапазона).

Индекс, передаваемый объекту *TCOLLECTION* при работе в среде Turbo Vision, выходит за границы диапазона.

214 Collection overflow error (Переполнение коллекции).

Эта ошибка генерируется объектом *TCOLLECTION* при работе в среде Turbo Vision в случае, если делается попытка добавить элемент к коллекции, которую нельзя расширять.

П4. СТАНДАРТНЫЕ БИБЛИОТЕЧНЫЕ МОДУЛИ

П4.1. МОДУЛЬ SYSTEM

Модуль *SYSTEM* является основной библиотекой Турбо Паскаля. Он реализует подпрограммы для всех встроенных возможностей, таких как ввод/вывод, обработка строк, эмуляция арифметического сопроцессора, управление оверлеями и динамическое распределение памяти. Модуль *SYSTEM* используется автоматически любым модулем или программой и никогда не указывается в предложении *USES*.

П4.1.1. Константы

const

```
ErrorAddr   : Pointer = NIL; {Адрес ошибки времени выполнения}
ExitProc    : Pointer = NIL; {Процедура выхода}
ExitCode    : Integer = 0;   {Выходной код}
FileMode    : Byte      = 2; {Режим открытия файла}
FreeList    : Pointer = NIL; {Список свободных блоков кучи}
HeapEnd     : Pointer = NIL; {Конец кучи}
HeapError   : Pointer = NIL; {Функция обработки ошибок кучи}
HeapOrg     : Pointer = NIL; {Начало кучи}
HeapPtr     : Pointer = NIL; {Указатель кучи}
InOutRes    : Integer = 0;   {Буфер результата ввода/вывода}
OvrCodeList : Word      = 0; {Список оверлейных сегментов кода}
OvrDebugPtr : Pointer = NIL; {Адрес отладчика оверлеев}
OvrDosHandle: Word      = 0; {Обработчик DOS оверлеев}
OvrEmsHandle: Word      = 0; {Обработчик EMS оверлеев}
OvrHeapEnd  : Word      = 0; {Конец буфера оверлеев}
OvrHeapOrg  : Word      = 0; {Начало буфера оверлеев}
OvrHeapPrt  : Word      = 0; {Указатель буфера оверлеев}
OvrHeapSize : Word      = 0; {Начальный размер буфера оверлеев}
OvrLoadList : Word      = 0; {Список загруженных оверлеев}
RandSeed    : LongInt   = 0; {Опорное значение генератора случайных чисел}
PrefixSeg   : Word      = 0; {Префикс программного сегмента}
Seg0040     : Word      = $0040 {Сегмент $0040}
SegA000     : Word      = $A000 {Сегмент $A000}
SegB000     : Word      = $B000 {Сегмент $B000}
SegB800     : Word      = $B800 {Сегмент $B800}
SelectorInc : Word      = $1000 {Шаг наращивания селекторов}
StackLimit  : Word      = 0;   {Указатель на нижнюю границу стека}
Test8086    : Byte      = 0;   {Признак наличия процессора 80x86}
Test8087    : Byte      = 0;   {Признак наличия сопроцессора}
```

OvrCodeList, *OvrHeapSize*, *OvrDebugPtr*, *OvrHeapOrg*, *OvrHeapPtr*, *OvrHeapEnd*, *OvrHeapList*, *OvrDosHandle* и *OvrEmsHandle* используются модулем *Overlay* для реализации администратора оверлеев. Буфер оверлеев размещается между сегментом стека и кучей и *OvrHeapOrg* и *OvrHeapEnd* содержат адреса начала и конца сегмента. Размер буфера оверлеев по умолчанию соответствует размеру максимального оверлея в программе; если в программе нет оверлеев, то размер оверлейного буфера равен нулю.

HeapOrg, *HeapPtr*, *HeapEnd*, *FreeList*, *HeapError* используются администратором кучи для реализации распределения динамической памяти.

Переменные *ExitProc*, *ExitCode* и *ErrorAddr* реализуют процедуры выхода.

PrefixSeg содержит сегментную часть адреса префикса программного сегмента (PSP).

StackLimit содержит смещение от нижней границы сегмента стека, соответствующее наименьшему допустимому значению регистра SP. Обычно *StackLimit* равен нулю; если программа откомпилирована с опциями *{N+,E+}*, компилятор будет устанавливать его равным

224 для резервирования рабочего пространства на нижней границе сегмента стека, используемого для эмуляции сопроцессора.

InOutRes используется встроенными программами ввода/вывода для запоминания значения, возвращаемого при последующем вызове стандартной функции *IOResult*.

RandSeed содержит опорное число генератора случайных чисел. При присваивании конкретного значения этой переменной функция *Random* будет генерировать строго определенную последовательность псевдослучайных чисел. Процедура *Randomize* заносит в переменную *RandSeed* текущее системное время и тем самым обеспечивает новую псевдослучайную последовательность при очередном прогоне программы.

FileMode позволяет изменить режим доступа, с которым открываются типизированные и нетипизированные файлы стандартной процедурой *RESET*. Текстовые файлы этой процедурой открываются только для чтения.

Seg0040 хранит сегмент участка памяти, который MS-DOS использует для размещения своих переменных.

SegA000 содержит сегмент начала видеопамати для графического режима работы, а *SegB000* и *SegB800* — сегменты видеопамати для текстового режима.

SelectorInc используется как шаг наращивания т.н. селекторов — указателей, обеспечивающих линейную модель памяти.

Test8086 хранит результат теста автообнаружения процессора Intel 80x86, а *Test8087* — сопроцессора.

П4.1.2. Переменные

var

```
Input      : Text;           {Стандартный файл ввода}
Output     : Text;           {Стандартный файл вывода}
SaveInt00: Pointer;          {Сохранение вектора $00}
.....
SaveInt75: Pointer;          {Сохранение вектора $75}
```

Input и *Output* — стандартные файлы, необходимые в каждой реализации языка Паскаль. По умолчанию они ссылаются на стандартные входные и выходные файлы ДОС: *Input* — на клавиатуру, *Output* — на экран.

В момент загрузки среда Турбо Паскаля изменяет значения 18 векторов прерывания (векторы \$00, \$02, \$1N, \$21, \$23, \$24, \$35, \$36, \$37, \$38, \$39, \$3A, \$3B, \$3C, \$3D, \$3E, \$3F и \$75). В переменных *SaveIntXX* сохраняются старые значения этих векторов (XX — номер вектора: *SaveInt00* — для вектора \$00, *SaveInt1B* — для вектора \$1B и т.д.).

П4.1.3. Стандартные процедуры и функции

Процедуры управления

Procedure Break Обеспечивает немедленный выход из операторов повторения.

Procedure Continue Завершает очередной итерационный цикл операторов повторения.

Procedure Exit Позволяет немедленно выйти из текущей подпрограммы. При вызове из тела основной программы завершает ее работу.

Procedure Halt[(ExitCode: Word)] Останавливает выполнение программы и возвращает управление в операционную систему. Необязательный параметр *ExitCode* определяет код завершения программы.

Procedure RunError[(ErrorCode: Byte)] Останавливает выполнение программы и генерирует ошибку периода выполнения программы. Необязательный параметр *ErrorCode* определяет код ошибки.

Процедуры динамического распределения памяти

Procedure Dispose(var P: Pointer [,Destructor]) Уничтожает динамическую переменную, связанную с указателем *P*. Необязательный параметр *Destructor* определяет метод — деструктор для динамического объекта.

Procedure FreeMem(var P: Pointer; Size: Word) Уничтожает динамическую переменную *P* размером *Size* байт.

Procedure GetMem(var P: Pointer; Size: Word) Создает новую динамическую переменную *P* заданного размера *Size*.

Procedure Mark(var P: Pointer) Записывает в указателе *P* текущее состояние кучи.

Procedure New(var P: Pointer [,Constructor]) Создает новую динамическую переменную, связанную с указателем *P*. Необязательный параметр *Constructor* определяет метод — конструктор для динамического объекта.

Procedure Release(var P: Pointer) Возвращает кучу в состояние, которое было сохранено в указателе *P* процедурой *Mark*.

Функции динамического распределения памяти

Function MaxAvail: LongInt Возвращает размер наибольшего непрерывного свободного блока кучи, соответствующий размеру наибольшей динамической переменной, которая может быть помещена в кучу.

Function MemAvail: LongInt Возвращает количество имеющихся в куче свободных байт памяти.

Функции преобразования

Процедуры *Pack* и *UnPack*, определенные в стандартном Паскале, в Турбо Паскале не реализованы.

Function Chr(X: Byte): Char Возвращает символ с заданным порядковым номером *X*.

Function Ord(X): LongInt Возвращает порядковый номер, соответствующий значению *X* порядкового типа.

Function Round(R: Real): LongInt Округляет значение *R* вещественного типа до ближайшего целого.

Function Trunc(R: Real): LongInt Усекает значение вещественного типа до значения типа *LongInt* путем отбрасывания дробной части.

Арифметические функции

При компиляции в режиме использования сопроцессора или его эмуляции арифметические функции возвращают значение типа *EXTENDED*, в противном случае — типа *REAL*.

Function Abs(R: Real): Real Возвращает абсолютное значение аргумента.

Function ArcTan(R: Real): Real Возвращает арктангенс аргумента.

Function Cos(R: Real): Real Возвращает косинус аргумента.

Function Exp(R: Real): Real Возвращает экспоненту аргумента.

Function Frac(R: Real): Real Возвращает дробную часть аргумента.

Function Int(R: Real): Real Возвращает целую часть аргумента.

Function Ln(R: Real): Real Возвращает натуральный логарифм аргумента.

Function Pi: Real Возвращает значение числа $\pi = 3.1415926535897932385$.

Function Sin(R: Real): Real Возвращает синус аргумента.

Function Sqr(R: Real): Real Возвращает аргумент в квадрате.

Function Sqrt(R: Real): Real Возвращает квадратный корень аргумента.

Процедуры порядкового типа

Procedure Dec(var X [: DX: LongInt]) Уменьшает значение переменной X на величину DX, а если параметр DX не задан — на 1.

Procedure Inc(var X [: DX: LongInt]) Увеличивает значение переменной X на величину DX, а если параметр DX не задан — на 1.

Функции порядкового типа

Function Odd(X): Boolean Проверяет, является ли аргумент нечетным числом.

Function Pred(X) Возвращает предшествующее значение аргумента. Тип результата совпадает с типом аргумента.

Function Succ(X) Возвращает последующее значение аргумента. Тип результата совпадает с типом аргумента.

Строковые процедуры

Procedure Delete(var S: String; Index, Count: Integer) Удаляет Count символов из строки S, начиная с позиции Index.

Procedure Insert(SubS: String; var S: String; Index: Integer) Вставляет подстроку SubS в строку S, начиная с позиции Index.

Procedure Str(X [: Width [: Decimals]]; var S: String) Преобразует численное значение X в его строковое представление S.

Procedure Val(S: String; var X; var Code: Integer) Преобразует строковое значение S в его численное представление X. Параметр Code содержит признак ошибки преобразования (0 — нет ошибки).

Строковые функции

Function Concat(S1 [, S2, ..., SN]): String Выполняет конкатенацию последовательности строк.

Function Copy(S: String; Index, Count: Integer): String Возвращает подстроку из строки S, начиная с позиции Index и длиной Count символов.

Function Length(S: String): Byte Возвращает текущую длину строки S.

Function Pos(SubS, S: String): Byte Возвращает позицию, начиная с которой в строке S располагается подстрока SubS (0 — S не содержит SubS).

Функции для работы с указателями и адресами

Function Addr(X): Pointer Возвращает адрес заданного объекта X.

Function Assigned: (var P): Boolean Проверяет, хранит ли указатель P значение, отличное от NIL, и возвращает TRUE в этом случае.

Function CSeg: Word Возвращает текущее значение регистра CS.

Function DSeg: Word Возвращает текущее значение регистра DS.

Function Ofs(X): Word Возвращает смещение заданного объекта.

Function Ptr(Seg, Ofs: Word): Pointer Преобразует сегмент *Seg* и смещение *Ofs* в значение типа указатель.

Function Seg(X): Word Возвращает сегмент для заданного объекта *X*.

Function SPtr: Word Возвращает текущее значение регистра *SP*.

Function SSeg: Word Возвращает текущее значение регистра *SS*.

Процедуры разного назначения

Procedure Exclude (var S: set of T; I: T) Исключает элемент *T* из множества *S*.

Procedure FillChar(var X; Count: Word; Value) Заполняет заданное количество *Count* последовательных байт переменной *X* указанным значением *Value* (выражение любого порядкового типа).

Procedure Include (var S: set of T; I: T) Включает элемент *T* во множество *S*.

Procedure Move(var X, Y; Count: Word) Копирует заданное количество последовательных байт из источника *X* в переменную *Y*.

Procedure Randomize Инициализирует случайным значением (текущим системным временем) встроенный генератор псевдослучайных чисел.

Функции разного назначения

Function Hi(X: Word): Byte Возвращает старший байт аргумента *X*.

Function High(X) Возвращает максимальное значение порядкового типа.

Function Lo(X: Word): Byte Возвращает младший байт аргумента *X*.

Function Low(X) Возвращает минимальное значение порядкового типа.

Function ParamCount: Word Возвращает число параметров, переданных программе в командной строке (строке вызова).

Function ParamStr(N: Byte): String Возвращает *N*-ый параметр командной строки.

Function Random[(Range: Word)] Возвращает псевдослучайное число. Если параметр *Range* опущен, функция возвращает вещественное число в диапазоне от 0 до 1, если указан — целое число в диапазоне от 0 до *Range* - 1.

Function SizeOf(X): Word Возвращает число байт, занимаемых аргументом.

Function Swap(X) Производит перестановку старших и младших байт двухбайтного аргумента *X*. Тип функции соответствует типу аргумента.

Function UpCase(C: char): Char Преобразует латинскую букву в заглавную.

Процедуры ввода/вывода

Procedure Assign(var F; Name: String) Связывает внешний файл *Name* с файловой переменной *F*.

Procedure ChDir(S: String) Устанавливает текущий каталог.

Procedure Close(**var** F) Закрывает открытый файл.

Procedure Erase(**var** F) Удаляет внешний файл.

Procedure GetDir(D: Byte; **var** S: String) Возвращает каталог по умолчанию S на заданном диске D.

Procedure Mkdir(S: String) Создает подкаталог S.

Procedure Rename(**var** F) Переименовывает внешний файл.

Procedure Reset(**var** F) Открывает существующий файл для чтения или изменения.

Procedure Rewrite(**var** F) Создает и открывает новый файл.

Procedure Rmdir(S: String) Удаляет пустой подкаталог.

Procedure Seek(**var** F; N: LongInt) Устанавливает текущую позицию файла на указанный элемент (не используется с текстовыми файлами).

Procedure Truncate(**var** F) Усекает размер файла до текущей позиции в файле (не используется с текстовыми файлами)

Функции ввода/вывода

Function EOF(**var** F): Boolean Возвращает для файла F признак конца файла.

Function FilePos(**var** F): LongInt Возвращает текущую позицию в файле (не используется с текстовыми файлами)

Function FileSize(**var** F): LongInt Возвращает текущий размер файла (не используется с текстовыми файлами).

Function IOResult: Integer Возвращает целое значение, являющееся состоянием последней выполненной операции ввода/вывода.

Процедуры для текстовых файлов

Procedure Append(**var** F: Text) Открывает существующий файл для расширения.

Procedure Flush(**var** F: Text) Выталкивает буфер файла вывода.

Procedure Read([**var** F: Text;] V1 [, V2, ..., VN]) Считывает одно или более значений из текстового файла в одну или более переменных.

Procedure Readln Выполняет те же действия, что и Read, а потом делает пропуск до начала следующей строки файла.

Procedure SetTextBuf(**var** F: Text; **var** Buf [, Size: Word]) Назначает буфер ввода/вывода для текстового файла. Параметр Size определяет длину буфера в байтах (если Size опущен, длина буфера равна 128 байтам).

Procedure Write([**var** F: Text;] V1 [, V2, ..., VN]) Записывает в текстовый файл одно или более значений.

Procedure Writeln Выполняет те же действия, что и Write, а затем добавляет к файлу маркер конца строки.

Функции для текстовых файлов

Function Eoln(**var** F: Text): Boolean Возвращает признак конца строки.

Function SeekEof(**var** F: Text): Boolean Возвращает признак конца файла. Предварительно пропускает все пробелы, символы табуляции и признаки конца строк.

Function SeekEoln(**var** F: Text): Boolean Возвращает признак конца строки. В отличие от Eoln предварительно пропускает все пробелы и символы табуляции.

Процедуры для нетипизированных файлов

Procedure BlockRead(**var** F: File; **var** Buf; Count: Word [**var** Result: Word])
Считывает в переменную Buf Count записей из файла F. Необязательный параметр Result содержит истинное количество считанных записей.

Procedure BlockWrite(**var** F: File; **var** Buf; Count: Word [**var** Result: Word])
Передает Count записей из переменной Buf в файл F. Необязательный параметр Result содержит истинное количество переданных записей.

П4.2. МОДУЛЬ DOS

Модуль Dos реализует ряд очень полезных программ операционной системы и обработки файлов. Ни одна из программ модуля Dos не определена в стандарте Паскале и поэтому они размещены в собственном модуле.

П4.2.1. Константы

Константы регистра флагов

Следующие константы используются для проверки отдельных битов флага в регистре Flags после вызова Intr или MSDOS:

```
const
  FCarry      = $0001;  FAuxiliary = $0010;   FSign      = $0080;
  FParity     = $0004;  FZero      = $0040;   FOverflow  = $0800;
```

Константы режима доступа к файлу

```
const
  fmClosed = $D7B0; {Маска режима файл закрыт}
  fmInput  = $D7B1; {Маска режима открыт для чтения}
  fmOutput  = $D7B2; {Маска режима открыт для записи}
  fmInOut   = $D7B3; {Маска режима открыт для чтения и записи}
```

Константы атрибутов файла

```
const
  ReadOnly = $01;      {Маска только для чтения}
  Hidden   = $02;      {Маска скрытого файла}
  SysFile  = $04;      {Маска системного файла}
  VolumeID = $08;      {Маска заголовка тома}
  Directory = $10;     {Маска каталога}
  Archive  = $20;      {Маска архивного файла}
  AnyFile  = $3F;      {Маска любого файла}
```

П4.2.2. Типы

```
type
  {Типизированные и нетипизированные файлы}
  FileRec = record
    Handle : Word;
```

```

Mode      : Word;
RecSize   : Word;
Private   : array [1..6] of Byte;
UserData  : array [1..16] of Byte;
Name      : array [0..79] of Char;
end;
{Текстовые файлы}
TextBuf = array [0..127] of Char;
TextRec = record
  Handle   : Word;
  Mode     : Word;
  BufSize  : Word;
  Private  : Word;
  BufPos   : Word;
  BufEnd   : Word;
  BufPtr   : ^TextBuf;
  OpenFunc : Pointer;
  InOutFunc : Pointer;
  FlushFunc : Pointer;
  CloseFunc : Pointer;
  UserData : array [1..16] of Byte;
  Name     : array [0..79] of Char;
  Buffer    : TextBuf;
end;

Registers = record
  case Integer of
    0: (AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags: Word);
    1: (AL, AH, BL, BH, CL, CH, DL, DH: Byte);
  end;

DateTime = record
  Year, Month, Day, Hour, Min, Sec: Integer;
end;

SearchRec = record
  File: array [1..21] of Byte;
  Attr: Byte;
  Time: LongInt;
  Size: LongInt;
  Name: String [12];
end;

DirStr  = String [67];      {Диск и каталог}
NameStr = String [8];      {Имя файла}
ExtStr  = String [4];      {Расширение файла}
ComStr  = String [127];    {Командная строка}
PathStr = String [79];     {Полный маршрут поиска файла}

```

П4.2.3. Переменные

```

var
  DosError : Integer;
  Значение, запомненное в DosError, представляет собой код ошибки операционной системы:
  0 - нет ошибки                      8 - нет памяти
  2 - файл не найден                  10 - неправильная среда
  3 - путь не найден                  11 - неправильный формат
  5 - доступ запрещен                 18 - больше нет файлов
  6 - неверный обработчик

```

П4.2.4. Процедуры и функции

Процедуры даты и времени

Procedure GetDate(var Year, Month, Day, DayOfWeek: Word) Возвращает текущую дату.

Procedure GetFTime(var F; var Time: LongInt) Возвращает дату и время последнего обновления файла.

Procedure GetTime(var Hour, Minute, Second, Sec100: Word) Возвращает текущее время.

Procedure PackTime(var Pack: DateTime; var Time: LongInt) Преобразует запись *Pack* в 4-байтовое упакованное значение даты и времени типа *LongInt*, используемое процедурой *SetFTime*. Поля записи *DateTime* не проверяются на диапазон.

Procedure SetData(Year, Month, Day: Word) Устанавливает текущую дату.

Procedure SetFTime(var F; Time: LongInt) Устанавливает время и дату последнего обновления файла.

Procedure SetTime(Hour, Minute, Second, Sec100: Word) Устанавливает текущее время.

Procedure UnpackTime(Time: LongInt; var Pack: DateTime) Преобразует 4-байтовое упакованное значение даты и времени, возвращаемое *GetFTime*. *FindFirst* или *FindNext*, в распакованную запись типа *DateTime*.

Процедуры обслуживания прерываний

Procedure GetIntVec(IntNo: Byte; var Vec: Pointer) Возвращает адрес, хранящийся в указанном векторе прерывания

Procedure Intr(IntNo: Byte; var R: Registers) Выполняет указанное программное прерывание.

Procedure MSdos(var R: Registers) Выполняет функцию операционной системы.

Procedure SetIntVec(IntNo: Byte; P: Pointer) Устанавливает адрес для указанного вектора прерывания.

Функции статуса диска

Function DiskFree(Disk: Word): LongInt Возвращает число свободных байт на указанном диске;

Function DiskSize(Disk: Word): LongInt Возвращает полный объем указанного диска в байтах

Процедуры обработки файлов

Procedure FindFirst(Path: String; Attr: Byte; var Search: SearchRec) Ищет в указанном или текущем каталоге первый файл, соответствующий заданному имени файла и набору атрибутов.

Procedure FindNext(var Search: SearchRec) Возвращает следующий файл, соответствующий имени и атрибутам, указанным в предыдущем вызове *FindFirst*.

Procedure GetFAttr(var F; var Attr: Byte) Возвращает атрибуты файла.

Procedure SetFAttr(var F; Attr: Byte) Устанавливает атрибуты файла.

Procedure FSplit(Path: PathStr; var Dir: DirStr; var Name: NameStr; var Ext: ExtStr) Разбивает имя файла *Path* на 3 составные части (каталог, имя файла, расширение).

Функции обработки файла

Function FExpand(Name: PathStr): PathStr Берет имя файла *Name* и возвращает полное имя файла (устройство, каталог, имя и расширение).

Function FSearch(Name: PathStr; DirList: String): PathStr Ищет файл *Name* в списке *DirList* каталогов. Элементы списка разделяются точкой с запятой.

Процедуры обработки процессов

Procedure Exec(Name: PathStr; CmdLine: String) Выполняет заданную программу *Name* с указанной командной строкой *CmdLine*.

Procedure Keep(Code: Word) Завершает программу и оставляет ее резидентной в памяти.

Procedure SwapVectors Обменивает сохраненные векторы прерываний из переменных *SaveIntXX* с текущими векторами.

Функции управления программой

Function DosExitCode: Word Возвращает код завершения подпроцесса.

Функции управления средой

Function EnvCount: Integer Возвращает число переменных окружения, содержащихся в среде ДОО.

Function EnvStr(Index: Integer): String Возвращает указанную переменную окружения ДОО.

Function GetEnv(EnvVar: String): String Возвращает значение указанной переменной окружения ДОО.

Дополнительные процедуры

Procedure GetCBreak(var CBreak: Boolean) Возвращает состояние проверки Ctrl-Break.

Procedure SetCBreak(CBreak: Boolean) Устанавливает состояние проверки Ctrl-Break в ДОО.

Procedure GetVerify(var Verify: Boolean) Возвращает состояние флага верификации в ДОО.

Procedure SetVerify(Verify: Boolean) Устанавливает состояние флага верификации в ДОО.

Дополнительные функции

Function DosVersion: Word Возвращает номер версии ДОО.

П4.3. МОДУЛЬ CRT

Модуль *Crt* содержит подпрограммы управления текстовым выводом на экран дисплея, звуковым генератором и чтением клавиатуры.

В режиме текстового вывода используются следующие координаты экрана: левый верхний угол экрана имеет координаты 1,1; горизонтальная координата возрастает слева направо, вертикальная — сверху вниз. Если на экране определено окно, все координаты определяются относительно границ окна. Исключением являются координаты процедуры *Window* установки границ окна, которые всегда задаются относительно границ экрана.

Для чтения клавиатуры используются две функции — *KeyPressed* и *ReadKey*. Функция *KeyPressed* определяет факт нажатия на любую клавишу и не приостанавливает дальнейшее исполнение программы. Функция *KeyPressed* читает расширенный код нажатой клавиши. Если к моменту обращения к функции не была нажата ни одна клавиша, программа приостанавливает свою работу, ожидая действий пользователя.

Управление звуковым генератором строится по схеме *Sound — Delay — NoSound*. Процедура *Sound* включает звуковой генератор и заставляет его непрерывно генерировать звук нужного тона. Процедура *Delay* приостанавливает работу программы на заданное число миллисекунд реального времени. Процедура *NoSound* отключает звуковой генератор.

4.3.1. Константы

Константы режима работы

const		
BW40	= 0;	{Черно-белый, 40 символов, 25 строк}
BW80	= 2;	{Черно-белый, 80 x 25}
Mono	= 7;	{Монохромный, 80 x 25}
CO40	= 1;	{Цветной, 40 x 25}
CO80	= 3;	{Цветной, 80 x 25}
Font8x8	= 256;	{Для EGA/VGA режим 43 или 50 строк}
C40	= CO40;	{Для совместимости с версией 3.0}
C80	= CO80;	{Для совместимости с версией 3.0}

Константы цветов

const		
Black	= 0;	{Черный}
Blue	= 1;	{Синий}
Green	= 2;	{Зеленый}
Cyan	= 3;	{Голубой}
Red	= 4;	{Красный}
Magenta	= 5;	{Фиолетовый}
Brown	= 6;	{Коричневый}
LightGray	= 7;	{Светло-серый}
DarkGray	= 8;	{Темно-серый}
LightBlue	= 9;	{Ярко-синий}
LightGreen	= 10;	{Ярко-зеленый}
LightCyan	= 11;	{Ярко-голубой}
LightRed	= 12;	{Розовый}
LightMagenta	= 13;	{Малиновый}
Yellow	= 14;	{Желтый}
White	= 15;	{Белый}
Blink	= 128;	{Мерцание символа}

П4.3.2. Переменные

var		
CheckBreak	: Boolean;	{Разрешает/запрещает контроль Ctrl-Break}
CheckEof	: Boolean;	{Разрешает/запрещает контроль Ctrl-Z}

CheckSnow : Boolean	{Разрешает/запрещает контроль "снега"}
DirectVideo : Boolean;	{Разрешает/запрещает прямой доступ к видеопамати}
LastMode : Word;	{Хранит последний текстовый режим}
TextAttr : Byte;	{Хранит текущий байт атрибутов}
WindMin : Word;	{Координаты левого верхнего угла текущего окна}
WindMax : Word;	{Координаты правого нижнего угла}

4.3.3. Процедуры и функции

Функции

Function KeyPressed: Boolean Возвращает *True*, если на клавиатуре была нажата клавиша, и *False* в противном случае. Не задерживает исполнение программы.

Function ReadKey: char Читает символ с клавиатуры без эхоповтора на экране. Приостанавливает исполнение программы до нажатия на любую клавишу, кроме *Shift*, *Ctrl*, *Alt*, *CapsLock*, *NumLock*, *ScrollLock*.

Function WhereX: Byte Возвращает горизонтальную координату текущей позиции курсора относительно текущего окна.

Function WhereY: Byte Возвращает вертикальную координату текущей позиции курсора относительно текущего окна.

Процедуры

Procedure AssignCrt(var F: Text) Связывает с файловой переменной устройство *CON* (клавиатуру для ввода и экран для вывода).

Procedure ClrEol Удаляет все символы от текущей позиции курсора до конца строки без перемещения курсора.

Procedure ClrScr Очищает экран (окно) и помещает курсор в верхний левый угол.

Procedure Delay(D: Word) Приостанавливает работу программы на указанное число *D* миллисекунд.

Procedure DellLine Удаляет строку, на которой находится курсор, и перемещает все строки ниже этой строки на строку вверх. Нижняя строка очищается.

Procedure GotoXY(X, Y: Byte) Перемещает курсор в нужное место экрана (окна).

Procedure HighVideo Устанавливает высокую яркость символов.

Procedure InsLine Вставляет пустую строку в позицию курсора.

Procedure LowVideo Устанавливает низкую яркость символов.

Procedure NormVideo Устанавливает нормальную яркость символов.

Procedure NoSound Выключает звуковой генератор.

Procedure Sound(F: Word) Включает звуковой генератор. *F* — частота звука (Гц).

Procedure TextBackground(Color: Byte) Устанавливает цвет фона.

Procedure TextColor(Color: Byte) Устанавливает цвет символов.

Procedure TextMode(Mode: Word) Устанавливает нужный текстовый режим.

Procedure Window(X1, Y1, X2, Y2: Byte) Определяет текстовое окно на экране. *X1*, *Y1* — координаты левого верхнего угла, *X2*, *Y2* — правого нижнего угла.

П4.4. МОДУЛЬ GRAPH

Модуль *Graph* представляет собой мощную библиотеку графических подпрограмм универсального назначения, рассчитанную на работу с наиболее распространенными графическими адаптерами IBM-совместимых ПК. Подпрограммы модуля *Graph* обеспечивают различные режимы работы многорежимных адаптеров, полностью используют их цветовые возможности и разрешающую способность.

При исполнении графических программ требуется автономный драйвер графического адаптера (*BGI*-файл). Если программа использует штриховые шрифты, то кроме того нужен один или несколько шрифтовых файлов (*CHR*-файлы). При необходимости драйвер и шрифты могут быть включены в тело программы еще на этапе компиляции.

Графические драйверы поддерживают следующие графические адаптеры (и полностью совместимые с ними):

CGA	Hercules
MCGA	AT&T 400
EGA	3270 PC
VGA	IBM-8514

Для поддержки этих аппаратных средств используются следующие драйверы:

CGA.BGI	Драйвер для CGA, MCGA
EGAVGA.BGI	Драйвер для EGA, VGA
HERC.BGI	Драйвер для монохромного Hercules
ATT.BGI	Драйвер для AT&T 6300 (400 строк)
PC3270.BGI	Драйвер для IBM 3270 PC
IBM8514.BGI	Драйвер для IBM 8514

Во время выполнения программы процедура *InitGraph* автоматически распознает графический адаптер, установленный на ПК, загружает и инициализирует соответствующий графический драйвер, переводит адаптер в графический режим и возвращает управление вызывающей программе. Процедура *CloseGraph* выгружает драйвер из памяти и восстанавливает текстовый режим работы адаптера.

Подпрограммы модуля *Graph* позволяют адресоваться к любому элементу (пикселю) растрового графического экрана и управлять светимостью этого элемента. Для указания пикселя используется следующая система координат: верхний левый угол графического экрана имеет координаты 0,0; горизонтальная координата *X* увеличивается слева направо, вертикальная координата *Y* увеличивается сверху вниз. Например, в режиме 640x480 (адаптер VGA) правый нижний угол экрана имеет координаты 639, 479, а центр экрана — координаты 319, 239.

Некоторые графические подпрограммы используют понятие текущего указателя. Указатель содержит координаты того пикселя, начиная с которого будет строиться изображение подпрограммами *LineTo*, *LineRel*, *OutText* и др. В этом смысле указатель подобен текстовому курсору, но, в отличие от него, не имеет видимого изображения на экране.

Для вывода текстовых сообщений на графический экран модуль *Graph* предоставляет один матричный и 4 штриховых шрифтов. Каждый символ матричного шрифта на экране реализуется в виде матрицы из 8x8 пикселей. Штриховые шрифты для каждого символа определяют набор штрихов (векторов), с помощью которых на экране создается (вычерчивается) соответствующий символ. Штриховые шрифты позволяют изменять размеры текстовых надписей в широких пределах без существенного ухудшения качества изображения символов. Однако стандартные штриховые шрифты не содержат символы кириллицы.

В модуле *Graph* имеется несколько процедур для рисования элементарных графических фигур — точек, линий, окружностей, дуг и т.п. При необходимости замкнутые фигуры могут быть закраснены различными цветами и стилями (образцами закраски).

Процедура *SetViewPort* создает на экране графическое окно. Если окно определено, весь дальнейший графический вывод осуществляется относительно координат этого окна и отсекается его границами.

В модуле имеются средства сохранения и выдачи изображений, работы с несколькими графическими страницами, установки необходимых цветов.

4.4.1. Константы

Константы задания драйверов и режимов работы

```
const
Detect      = 0;
CGA         = 1;
MCGA        = 2;
EGA         = 3;
EGA64       = 4;
EGAMono     = 5;
IBM8514     = 6;
HercMono    = 7;
ATT400      = 8;
VGA         = 9;
PC3270      = 10;
CurrentDriver=-128; {Передается в GetModeRange}
CGAC0       = 0;    {320x200; палитра 0; 1 страница}
CGAC1       = 1;    {320x200; палитра 1; 1 страница}
CGAC2       = 2;    {320x200; палитра 2; 1 страница}
CGAC3       = 3;    {320x200; палитра 3; 1 страница}
CGAHi       = 4;    {640x200; 1 страница}
MCGAC0      = 0;    {320x200; палитра 0; 1 страница}
MCGAC1      = 1;    {320x200; палитра 1; 1 страница}
MCGAC2      = 2;    {320x200; палитра 2; 1 страница}
MCGAC3      = 3;    {320x200; палитра 3; 1 страница}
MCGAMed     = 4;    {640x200; 1 страница}
MCGAHi      = 5;    {640x480; 1 страница}
EGALo       = 0;    {640x200; 16 цветов; 4 страницы}
EGAHi       = 1;    {640x350; 16 цветов; 2 страницы}
EGA64Lo     = 0;    {640x200; 16 цветов; 1 страница}
EGA64Hi     = 1;    {640x350; 4 цвета; 1 страница}
EGAMonoHi   = 3;    {640x350; 64K: 1 страница; 256K: 2 страницы}
HercMonoHi  = 0;    {720x348; 2 страницы}
ATT400C0    = 0;    {320x200; палитра 0; 1 страница}
ATT400C1    = 1;    {320x200; палитра 1; 1 страница}
ATT400C2    = 2;    {320x200; палитра 2; 1 страница}
ATT400C3    = 3;    {320x200; палитра 3; 1 страница}
ATT400Med   = 4;    {640x200; 1 страница}
ATT400Hi    = 5;    {640x400; 1 страница}
VGA Lo      = 0;    {640x200; 16 цветов; 4 страницы}
VGAMed      = 1;    {640x350; 16 цветов; 2 страницы}
VGAHi       = 2;    {640x480; 16 цветов; 1 страница}
PC3270Hi    = 0;    {720x350; 1 страница}
IBM8514Lo   = 0;    {640x480; 256 цветов}
IBM8514Hi   = 1;    {1024x768; 256 цветов}
```

Значения ошибок, возвращаемые GraphResult

```
const
grOk        = 0;    {Нет ошибок}
grNoInitGraph = -1;  {Графика не инициализирована}
grNotDetected = -2;  {Графическое устройство не обнаружено}
grFileNotFound = -3; {Файл драйвера устройства не найден}
grInvalidDriver=-4;  {Неправильный файл драйвера устройства}
grNoLoadMem  = -5;   {Нет памяти для загрузки драйвера}
grNoScanMem  = -6;   {Нет памяти для просмотра областей}
grNoFloodMem = -7;   {Нет памяти для закраски областей}
grFontNotFound = -8; {Файл шрифта не найден}
```

```

grNoFontMem      = -9;      {Нет памяти для загрузки шрифта}
grInvalidMode    = -10     {Недопустимый графический режим}
grError          = -11     {Общая ошибка}
grIOerror        = -12     {Ошибка графического ввода/вывода}
grInvalidFont    = -13     {Неверный файл шрифта}
grInvalidFontNum = -14     {Неверный номер шрифта}

```

Константы цвета

```

const
  Black      = 0;      {Черный}
  Blue       = 1;      {Синий}
  Green      = 2;      {Зеленый}
  Cyan       = 3;      {Голубой}
  Red        = 4;      {Красный}
  Magenta    = 5;      {Фиолетовый}
  Brown      = 6;      {Коричневый}
  LightGray  = 7;      {Светло-серый}
  DarkGray   = 8;      {Темно-серый}
  LightBlue  = 9;      {Ярко-синий}
  LightGreen = 10;     {Ярко-зеленый}
  LightCyan  = 11;     {Ярко-голубой}
  LightRed   = 12;     {Розовый}
  LightMagenta = 13;   {Малиновый}
  Yellow     = 14;     {Желтый}
  White      = 15;     {Белый}

```

Следующие константы цветов могут быть использованы с *SetRGBPalette* для выбора цветов на графическом адаптере IBM 8514:

```

const
  EGABlack      = 0; {Темные цвета}
  EGABlue       = 1;
  EGAGreen      = 2;
  EGACyan       = 3;
  EGARed        = 4;
  EGAMagenta    = 5;
  EGALightGray  = 7;
  EGABrown      = 20;
  EGADarkGray   = 56; {Светлые цвета}
  EGALightBlue  = 57;
  EGALightGreen = 58;
  EGALightCyan  = 59;
  EGALightRed   = 60;
  EGALightMagenta = 61;
  EGAYellow     = 62;
  EGAWhite      = 63;

```

Константы типов и толщины линий

```

const
  SolidLn      = 0;      {Сплошная}
  DottedLn     = 1;      {Точечная}
  CenterLn     = 2;      {Штрихпунктирная}
  DashedLn     = 3;      {Пунктирная}
  UserBitLn    = 4;      {Тип определяется пользователем}
  NormWidth    = 1;      {Нормальная толщина}
  ThickWidth   = 3;      {Тройная толщина}

```

Константы управления шрифтом

```
const
  DefaultFont = 0;    {Матричный шрифт 8x8}
  TriplexFont = 1;    {Шрифт триплекс; файл TRIP.CHR}
  SmallFont   = 2;    {Мелкий шрифт; файл LITT.CHR}
  SansSerifFont= 3;   {Прямой шрифт; файл SANS.CHR}
  GothicFont  = 4;    {Готический шрифт; файл GOTH.CHR}
  HorizDir    = 0;    {Горизонтальное направление}
  VertDir     = 1;    {Вертикальное направление}
  UserCharSize= 0;    {Размер символа, определяемый пользователем}
  LeftText    = 0;    {Указатель слева от текста}
  CenterText  = 1;    {Указатель по центру текста}
  RightText   = 2;    {Указатель справа от текста}
  BottomText  = 0;    {Указатель снизу от текста}
  CenterText  = 1;    {Указатель по центру текста}
  TopText     = 2;    {Указатель сверху от текста}
```

Константы отсечения

```
const
  ClipOn = True;      {Отсекать изображение}
  ClipOff= False;     {Игнорировать окно (не отсекай изображение)}
```

Константы для Bar3D

```
const
  TopOn = True;        {Рисовать вершину}
  TopOff= False;       {Не рисовать вершину}
```

Константы шаблона штриховки

```
const
  EmptyFill    = 0;    {Нет штриховки}
  SolidFill    = 1;    {Сплошная штриховка}
  LineFill     = 2;    {--- штриховка}
  LtSlashFill  = 3;    {/// штриховка}
  SlashFill    = 4;    {/// штриховка (толстыми линиями)}
  BkSlashFill  = 5;    {\ \ \ штриховка (толстыми линиями)}
  LtBkSlashFill= 6;    {\ \ \ штриховка}
  HatchFill    = 7;    {+++ штриховка}
  XHatchFill   = 8;    {xxx штриховка}
  InterleaveFill= 9;   {Штриховка в клетку}
  WideDotFill  =10;    {Штриховка редкими точками}
  CloseDotFill =11;    {Штриховка частыми точками}
  UserFill     =12;    {Штриховка задается пользователем}
```

Битовые константы

```
const
  CopyPut = 0;    {MOV}
  XORPut  = 1;    {XOR}
  OrPut   = 2;    {OR}
  AndPut  = 3;    {AND}
  NotPut  = 4;    {NOT}
```

Константа палитры

```
const
```

MaxColors = 15; {Максимальное количество цветов}

4.4.2. Типы

type

```

PaletteType = record            {Используется в GetPallette}
    Size    : Byte;
    Colors : array [0..MaxColors] of ShortInt;
end;
LineSettingsType = record    {Используется в GetLineSettings}
    LineStyle : Word;
    Pattern   : Word;
    Thickness : Word;
end;
TextSettingsType = record    {Используется в GetTextSettings}
    Font       : Word;
    Direction : Word;
    CharSize   : Word;
    Horiz      : Word;
    Vert       : Word;
end;
FillSettingsType = record    {Используется в GetFillSettings}
    Pattern : Word;
    Color   : Word;
end;
FillPatternType = array [1..8] of Byte;
PointType = record            {Для задания координат многоугольников}
    X,Y: Integer;
end;
ViewPortType = record        {Используется в GetViewSettings}
    X1, Y1, X2, Y2: Integer;
    Clip        : Boolean;
end;
ArcCoordsType = record        {Используется в GetArcCoords}
    X, Y        : Integer;
    Xstart, Ystart: Integer;
    Xend, Yend   : Integer;
end;

```

4.4.3. Переменные

var

```

GraphGetMemPrt : Pointer;    {Распределение кучи}
GraphFreeMemPrt: Pointer;    {Освобождение кучи}

```

Эти переменные указывают на программы управления кучей модуля *Graph*. Если Ваша программа использует собственный алгоритм управления памятью, присвойте адреса Ваших программ распределения и освобождения памяти переменным *GraphGetMemPrt* и *GraphFreeMemPrt*.

4.4.4. Процедуры

Procedure Arc(X, Y: Integer; StAngle, EndAngle, Radius: Word) Рисует дугу радиусом *Radius* от начального угла *StAngle* к конечному *EndAngle*, используя X,Y как координаты центра.

Procedure Bar(X1, Y1, X2, Y2: Integer) Рисует полосу заданного размера, используя текущий стиль и цвет.

Procedure Bar3D(X1, Y1, X2, Y2: Integer; Depth: Word; Top: Boolean) Рисует трехмерную полосу, используя текущий стиль и цвет.

Procedure Circle(X, Y: Integer; R: Word) Рисует окружность радиуса R, используя X,Y как координаты центра.

Procedure ClearDevice Очищает экран.

Procedure ClearViewport Очищает окно.

Procedure CloseGraph Закрывает графический режим.

Procedure DetectGraph(var Driver: Integer; Mode: Integer) Возвращает тип Driver установленного драйвера и текущий режим Mode его работы.

Procedure DrawPoly(NumPoints: Word; var PolyPoints) Рисует многоугольник из NumPoints вершин с координатами в PolyPoints текущим цветом и типом линии.

Procedure Ellipse(X, Y: Integer; StAngle, EndAngle, XR, YR: Word) Рисует эллиптическую дугу от начального угла StAngle к конечному углу EndAngle, используя X,Y как координаты центра и XR, YR — как горизонтальный и вертикальный радиусы.

Procedure FillEllipse(X, Y: Integer; XR, YR: Word) Рисует заштрихованный эллипс, используя X,Y как центр и XR, YR как горизонтальный и вертикальный радиусы.

Procedure FillPoly(NumPoints: Word; var PolyPoints) Рисует и штрихует многоугольник, содержащий NPoints вершин с координатами в PolyPoints.

Procedure FloodFill(X, Y: Integer; Border: Word) Штрихует замкнутую область, содержащую внутреннюю точку с координатами X,Y и ограниченную линией с цветом Border. Используется текущий образец штриховки и цвет.

Procedure GetArcCoords(var ArcCoo: ArcCoordsType) Возвращает координаты центра, начала и конца дуги.

Procedure GetAspectRatio(var XAsp, YAsp: Word) Возвращает два числа, позволяющие оценить отношение сторон графического экрана (XAsp/YAsp).

Procedure GetDefaultPalette(var Palette: PaletteType) Возвращает текущую палитру в записи PaletteType.

Procedure GetFillPattern(var FillPatt: FillPatternType) Возвращает текущий образец штриховки.

Procedure GetFillSettings(var FillInfo: FillSettingsType) Возвращает текущий образец и цвет штриховки.

Procedure GetImage(X1, Y1, X2, Y2: Integer; var BitMap) Сохраняет в переменной BitMap битовый образ указанной части экрана.

Procedure GetLineSettings(var LineInfo: LineSettingsType) Возвращает текущий стиль, шаблон и толщину линии.

Procedure GetModeRange(GraphDriver: Integer; var LoMode, HiMode: Integer) Для графического драйвера GraphDriver возвращает диапазон возможных режимов работы.

Procedure GetPalette(var Palette: PaletteType) Возвращает текущую палитру и ее размер.

Procedure GetTextSettings(var TextInfo: TextSettingsType) Возвращает текущий шрифт, направление, размер и выравнивание текста, установленные процедурами SetTextStyle и SetTextJustify.

Procedure GetViewSettings(var ViewPort: NiewPortType) Возвращает координаты и признак отсечки текущего окна.

Procedure GraphDefaults Устанавливает стандартные параметры графического режима.

Procedure InitGraph(var Driver, Mode: Integer; Path: String) Инициализирует графический режим. Переменные *Driver* и *Mode* должны содержать тип графического драйвера и его режим работы. Допускается указать *Driver* = 0 для автоматического определения этих параметров по результатам тестирования аппаратуры. Параметр *Path* определяет маршрут поиска файла графического драйвера.

Procedure Line(X1, Y1, X2, Y2: Integer) Рисует линию от точки *X1,Y1* до точки *X2,Y2*.

Procedure LineRel(DX, DY: Integer) Рисует линию от текущего указателя к точке, заданной приращением координат.

Procedure LineTo(X, Y: Integer) Рисует линию от текущего указателя к точке *X,Y*.

Procedure MoveRel(DX, DY) Смещает текущий указатель к точке, заданной приращением координат.

Procedure MoveTo(X, Y: Integer) Смещает текущий указатель к точке *X,Y*.

Procedure OutText(TextString: String) Выводит текстовую строку на экран.

Procedure OutTextXY(X, Y: Integer; TextString: String) Выводит текст в заданное место экрана.

Procedure PieSlice(X, Y: Integer; StAngle, EndAngle, Radius: Word) Рисует и штрихует сектор окружности радиусом *Radius* с центром в *X,Y* от начального угла *StAngle* к конечному углу *EndAngle*.

Procedure PutImage(X, Y: Integer; var BitMap; BitBlt: Word) Выводит битовый образ на экран.

Procedure PutPixel(X, Y: Integer; Color: Word) Выводит точку цветом *Color* с координатами *X,Y*.

Procedure Rectangle(X1, Y1, X2, Y2: Integer) Рисует прямоугольник, используя текущий цвет и тип линии.

Procedure RestoreCRTMode Восстанавливает текстовый режим работы экран.

Procedure Sector(X, Y: Integer; StAngle, EndAngle, XR, YR: Word) Рисует и штрихует сектор эллипса радиусами *XR,YR* с центром в *X,Y* от начального угла *StAngle* к конечному углу *EndAngle*.

Procedure SetActivePage(Page: Word) Устанавливает активную страницу для графического вывода.

Procedure SetAllPalette(var Palette) Изменяет все цвета палитры.

Procedure SetAspectRatio(XAsp, YAsp: Word) Изменяет масштабный коэффициент отношения сторон графического экрана.

Procedure SetBkColor(Color: Word) Устанавливает цвет фона.

Procedure SetColor(Color: Word) Устанавливает основной цвет, которым будет осуществляться рисование.

Procedure SetFillPattern(Pattern: FillPatternType; Color: Word) Устанавливает произвольный образец штриховки.

Procedure SetFillStyle(Pattern, Color: Word) Устанавливает образец штриховки и цвет.

Procedure SetGraphBufSize(Size: Word) Позволяет изменить размер буфера для функций штриховки.

Procedure SetGraphMode(Mode: Integer) Устанавливает новый графический режим и очищает экран.

Procedure SetLineStyle(LineStyle, Pattern, Thickness: Word) Устанавливает толщину и стиль линии.

Procedure SetPalette(ColorNum, Color: Word) Заменяет цвет палитры с номером *ColorNum* на цвет *Color*.

Procedure SetRGBPalette(ColorNum, Red, Green, Blue: Integer) Позволяет модифицировать палитру для IBM 8514 и VGA.

Procedure SetTextJustify(Horiz, Vert: Word) Устанавливает выравнивание текста, используемое в процедурах *OutText* и *OutTextXY*.

Procedure SetTextStyle(Font, Direction, CharSize: Word) Устанавливает текущий шрифт, стиль и размер текста.

Procedure SetUserCharSize(MultX, DivX, MultY, DivY: Word) Изменяет пропорции шрифта.

Procedure SetViewport(X1, Y1, X2, Y2: Integer; ClipOn: Boolean) Устанавливает текущее окно для графического вывода.

SetVisualPage(PageNo: Word) Устанавливает номер видимой графической страницы.

SetWriteMode(WriteMode: Integer) Устанавливает режим вывода (копирование или XOR) для линий, рисуемых процедурами *DrawPoly*, *Line*, *LineRel*, *LineTo*, *Rectangle*.

4.4.5. Функции

Procedure GetBkColor: Word Возвращает текущий фоновый цвет.

Procedure GetColor: Word Возвращает текущий цвет.

Procedure GetDriverName: String Возвращает строку с именем текущего драйвера.

Procedure GetGraphMode: Integer Возвращает текущий графический режим.

Procedure GetMaxColor: Word Возвращает максимальный цвет, который можно задать в *SetColor*.

Procedure GetMaxMode: Integer Возвращает номер максимального режима текущего загруженного драйвера.

Procedure GetMaxX: Integer Возвращает максимальную горизонтальную координату графического экрана.

Procedure GetMaxY: Integer Возвращает максимальную вертикальную координату графического экрана.

Procedure GetModeName(ModeNum: Word): String Возвращает строку с именем указанного графического режима.

Procedure GetPaletteSize: Integer Возвращает размер таблицы палитры.

Procedure GetPixel(X, Y: Integer): Word Возвращает цвет пикселя с координатами *X, Y*.

Procedure GetX: Integer Возвращает координату *X* текущего указателя.

Procedure GetY: Integer Возвращает координату *Y* текущего указателя.

Procedure GraphErrorMsg(ErrorCode: Integer): String Возвращает строку сообщения об ошибке для заданного кода *ErrorCode*.

Procedure GraphResult: Integer Возвращает код ошибки для последней графической операции.

Procedure ImageSize(X1, Y1, X2, Y2: Integer) Возвращает число байт, требуемое для сохранения прямоугольной области экрана.

Procedure InstallUserDriver(Name: String; AutoDetectPtr: Pointer): Integer Устанавливает пользовательский драйвер устройства в таблицу драйверов устройств.

Procedure InstallUserFont(FontFileName: String): Integer Устанавливает новый шрифт, который не встроен в BGI систему.

Procedure RegisterBGIDriver(Driver: Pointer): Integer Регистрирует драйвер для графической системы.

Procedure RegisterBGIfont(Font: Pointer): Integer Регистрирует шрифт для графической системы.

Procedure TextHeight(TextStr: String): Word Возвращает высоту строки в пикселах.

Procedure TextWidth(TextStr: String): Word Возвращает ширину строки в пикселах.

П5. ТЕКСТЫ ПРОГРАММ

П5.1. ПРОГРАММА ОПРЕДЕЛЕНИЯ ДНЯ НЕДЕЛИ

{Эта программа вводит дату в формате ДД ММ ГГГГ и выводит на экран соответствующий этой дате день недели. Описание программы см. п.2.7.1.}

```
var
  IsCorrectDate: Boolean; {Признак правильной даты}
  d,m,y         : Integer; {Вводимая дата - день, месяц и год}
  {-----}
Procedure InputDate(var d,m,y : Integer; var correctly : Boolean);
{Вводит в переменные d, m и y очередную дату и проверяет ее.
Если дата правильная, устанавливает correctly=true, иначе correctly=false }
begin {InputDate}
  Write('Введите дату в формате ДД ММ ГГГГ: ');
  ReadLn(d,m,y);
  correctly := (d>=1) and (d<=31) and (m>=1)
               and (m<=12) and (y>=1582) and (y<=4903)
end; {InputDate}
  {-----}
Procedure WriteDay(d,m,y : Integer);
const
  Days_of_week: array [0..6] of String [11] =
    ('воскресенье','понедельник','вторник',
     'среда','четверг','пятница','суббота');
var
  c, w : Integer;
begin
  if m < 3 then
    begin                                     {Месяц январь или февраль}
      m := m + 10;
      y := y - 1
    end
  else
      m := m - 2;                             {Остальные месяцы}
      c := y div 100;                         {Вычисляем столетие}
      y := y mod 100;                         {Находим год в столетии}
      w := abs(trunc(2.6*m-0.2)+d+y div 4+y+c div 4-2*c) mod 7;
      WriteLn(Days_of_week[w])
  end;
  {-----}
begin
  repeat
    InputDate(d,m,y,IsCorrectDate);
    if IsCorrectDate then
      WriteDay(d,m,y)
    until not IsCorrectDate
end.
```

П5.2. ОПРЕДЕЛЕНИЕ БИОРИТМОВ

{Программа для определения физической, эмоциональной и интеллектуальной активности человека. Вводится дата рождения и текущая дата. Программа вычисляет и выводит на экран общее количество дней, часов, минут и секунд, разделяющих обе даты, а также прогнозирует на месяц вперед даты, соответствующие максимуму и минимуму биоритмов. Описание программы см. п.2.7.2.}

```

const
  Size_of_Month: array [1..12] of Byte =
    (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
var
  d0, d,           {Дни рождения и текущий}
  m0, m,           {Месяцы рождения и текущий}
  y0, y,           {Годы рождения и текущий}
  dmin,            {Наименее благоприятный день}
  dmax,            {Наиболее благоприятный день}
  days: Integer;   {Количество дней от рождения}
  {-----}
Procedure InputDates(var d0,m0,y0,d,m,y : Integer);
{Вводит дату рождения и текущую дату. Контролирует правильность дат и их
непротиворечивость (текущая дата должна быть позже даты рождения)}
var
  correctly: Boolean; {Признак правильного ввода}
  {-----}
Procedure InpDate(text: String; var d,m,y: Integer);
{Выводит приглашение TEXT, вводит дату в формате ДД ММ ГГГГ и
проверяет ее правильность}
const
  YMIN = 1800;      {Минимальный правильный год}
  YMAX = 2000;      {Максимальный правильный год}
begin {InpDate}
  repeat
    Write(text);
    ReadLn(d,m,y);
    correctly := (y >= YMIN) and (Y <= YMAX) and (m >= 1)
                  and (m <= 12) and (d > 0);

    if correctly then
      if (m = 2) and (d = 29) and (y mod 4 = 0)
      then
        {Ничего не делать: это 29 февраля високосного года!}
      else
        correctly := d <= Size_of_Month[m];
      if not correctly then
        WriteLn('Ошибка в дате!')
      until correctly
end; {InpDate}
  {-----}
begin {InputDates}
  repeat
    InpDate(' Введите дату рождения в формате ДД ММ ГГГГ:', d0, m0, y0);
    InpDate(' Введите текущую дату: ', d, m, y);
    {Проверяем непротиворечивость дат:}
    correctly := y > y0;
    if not correctly and (y = y0) then
      begin
        correctly := m > m0;
        if not correctly and (m = m0) then
          correctly := d >= d0
        end
      until correctly
    end; {InputDates}
  {-----}
Procedure Get_numbers_of_days(d0,m0,y0,d,m,y: Integer; var days: Integer);
{Определение полного количества дней, прошедших от одной даты до другой }
  {-----}
procedure Variant2;

```

{Подсчет количества дней в месяцах, разделяющих обе даты }

```

var
  mm : Integer;
begin {Variant2}
  mm := m0;
  while mm < m do
    begin
      days := days + Size_of_Month[mm];
      if (mm = 2) and (y0 mod 4 = 0) then
        inc(days);
      inc(mm)
    end
  end; {Variant2}
  {-----}
Procedure Variant3;
{Подсчет количества дней в месяцах и годах, разделяющих обе даты}
var
  mm, yy : Integer;
begin {Variant3}
  mm := m0 + 1;
  while mm <= 12 do
    {Учитываем остаток года рождения:}
    begin
      days := days + Size_of_Month[mm];
      if (mm = 2) and (y0 mod 4 = 0) then
        inc(days);
      inc(mm)
    end;
  yy := y0 + 1;
  while yy < y do
    {Прибавляем разницу лет:}
    begin
      days := days + 365;
      if yy mod 4 = 0 then
        inc(days);
      inc(yy)
    end;
  mm := 1;
  while mm < m do
    {Прибавляем начало текущего года:}
    begin
      days := days + Size_of_Month[mm];
      if (y mod 4 = 0) and (mm = 2) then
        inc(days);
      inc(mm)
    end
  end; {Variant3}
  {-----}
begin {Get_numbers_of_days}
  if (y = y0) and (m = m0) then
    {Даты отличаются только днями:}
    days := d - d0
  else
    {Даты отличаются не только днями:}
    begin
      days := d + Size_of_Month[m0] - d0;
      {Учитываем количество дней в текущем месяце и количество дней до конца
        месяца рождения}
      if (y0 mod 4 = 0) and (m0 = 2) then
        inc(days);
      {Учитываем високосный год}
      if y = y0 then
        Variant2
      {Разница в месяцах одного и того же года}
      else
        Variant3
      {Даты отличаются годами}
    end
  end;
end;

```

```

    end
end; {Get_numbers_of_days}
{-----}
Procedure FindMaxMin(var dmin, dmax: Integer; days: Integer);
{Поиск критических дней}
const
    TF = 2*3.1416/23.6884;      {Период физической активности}
    TE = 2*3.1416/28.4261;      {Период эмоциональной активности}
    TI = 2*3.1416/33.1638;      {Период интеллектуальной активности}
    INTERVAL = 30;              {Интервал прогноза}
var
    min,                        {Накапливает минимум биоритмов}
    max,                        {Накапливает максимум биоритмов}
    x : Real;                   {Текущее значение биоритмов}
    i : Integer;
begin {FindMaxMin}
    max := sin(days*TF)+sin(days*TE)+sin(days*TI);
    min := max;                 {Начальное значение минимума и максимума
                                равно значению биоритмов для текущего дня}

    dmin := days;
    dmax := days;
    for i := 0 to INTERVAL do
        begin
            x := sin((days+i)*TF) + sin((days+i)*TE) +
                sin((days+i)*TI);
            if x > max then
                begin
                    max := x;
                    dmax := days + i
                end
            else if x < min then
                begin
                    min := x;
                    dmin := days + i
                end
            end
        end;
    end; {FindMaxMin}
{-----}
Procedure WriteDates(dmin,dmax,days : Integer);
{Определение и вывод дат критических дней. Вывод дополнительной информации о
числе прожитых дней, часов, минут и секунд }
{-----}
Procedure WriteDate(text: String; dd: Integer);
{Определение даты для дня DD от момента рождения. В глобальных переменных d,
m и y имеется текущая дата, в переменной DAYS - количество дней, прошедших от
момента рождения до текущей даты. Выводится сообщение ТЕХТ и найденная дата в
формате ДД-МЕС-ГГГГ}
const
    Names_of_Monthes : array [1..12] of String [3] =
        ('янв','фев','мар','апр','мая','июн',
         'июл','авг','сен','окт','ноя','дек');
var
    d0,m0,y0,ddd : Integer;
begin {WriteDate}
    d0 := d;
    m0 := m;
    y0 := y;
    ddd := days;
    while ddd < dd do
        begin

```



```

i,j : Integer;
begin {ShowField}
  for i := 1 to nrow do
    begin
      GotoXY(X0,i+4);
      write(i);                                {Номер ряда}
      GotoXY(X1,i+4);
      write(col[i]:2);                          {Количество фишек в ряду}
      for j := 1 to ncol[i] do                {Вывод ряда фишек:}
        begin
          GotoXY(X+2*j,i+4);
          if j<=col[i] then write(FISH) else write('.')
        end
      end
    end; {ShowField}
  {-----}
  Procedure Prepare;
  { Подготовка данных и формирование экрана }
  const
    Header0 = 'И Г Р А   Н И М';
    Header1 = 'Вы можете взять любое число фишек из любого ряда.';
    Header2 = 'Выигрывает тот, кто возьмет последнюю фишку.';
    Header3 = 'Номер ряда';
    Header4 = 'Кол-во фишек';
  var
    i : Integer;
  begin {Prepare}
    ClrScr;                                {Очищаем экран }
    {Выводим заголовок:}
    GotoXY((80-Length(Header0)) div 2,1);
    write(Header0);
    GotoXY((80-Length(Header1)) div 2,2);
    write(Header1);
    GotoXY((80-Length(Header2)) div 2,3);
    writeln(Header2);
    write(Header3);
    GotoXY(80-Length(Header4),4);
    write(Header4);
    {Подготовить начальную раскладку:}
    for i := 1 to nrow do col[i] := ncol[i]
  end; {Prepare}
  {-----}
  Procedure GetPlayerMove;
  { Получить, проконтролировать и отобразить ход игрока }
  const
    TEXT1 = 'Введите Ваш ход в формате РЯД КОЛИЧ '+'
              '(например, 2 3 - взять из 2 ряда 3 фишки)';
    TEXT2 = 'или введите 0 0 для выхода из игры; -1 0 для настройки игры';
    TEXT3 = 'Ваш ход:                ';
    Y      = 20;                        {номер строки для вывода сообщений}
  var
    correctly : Boolean;                {признак правильности сделанного хода}
    x1,x2      : Integer;              {вводимый ход}
  {-----}
  Procedure GetChange;
  { Ввести новую настройку игры (количество рядов и количество фишек в каждом
  ряду}
  const
    t1 = 'НАСТРОЙКА   И Г Р Ы';
    t2 = '(ввод количества рядов и количества фишек в каждом ряду)';

```



```

var
  correctly : Boolean;
  i         : Integer;
begin {GetChange}
  clrscr;
  GotoXY((80-Length(t1)) div 2,1);
  write(t1);
  GotoXY((80-Length(t2)) div 2,2);
  write(t2);
  repeat
    GotoXY(1,3);
    write('Введите количество рядов (максимум ',MAXROW,'):      ');
    GotoXY(WhereX-6,WhereY);
    readln(nrow);
    correctly := (nrow<=MAXROW) and (nrow>1);
    if not correctly then
      write(#7)
  until correctly;
  for i := 1 to nrow do
    repeat
      GotoXY(1,i+3);
      write(' ряд ',i,' , количество фишек (максимум ',MAXCOL,'):      ');
      GotoXY(WhereX-6,WhereY);
      readln(ncol[i]);
      correctly := (ncol[i]<=MAXCOL) and (ncol[i]>0);
      if not correctly then
        write(#7)
    until correctly
  end; {GetChange}
  {-----}
begin {GetPlayerMove}
  ShowField;           {Показать начальное состояние поля }
  { Сообщить игроку правила ввода хода: }
  GotoXY((80-Length(TEXT1)) div 2,Y);
  write(TEXT1);
  GotoXY((80-Length(TEXT2)) div 2,Y+1);
  write(TEXT2);
  repeat
    { Пригласить игрока ввести ход: }
    GotoXY(1,Y+2);
    Write(TEXT3);       {вывести приглашение и стереть предыдущий ход}
    GotoXY(WhereX-16,Y+2); {курсор влево на 16 позиций}
    ReadLn(x1,x2);      {ввести очередной ход}
    exit  := x1=0;       {контроль команды выхода}
    change := x1=-1;     {контроль команды изменения}
    if not (exit or change) then
      begin
        correctly := (x1>0) and (x1<=nrow) and
                      (x2<=col[x1]) and (x2>0);
        if correctly then
          begin
            col[x1] := col[x1]-x2; {изменить раскладку фишек}
            ShowField             {показать поле}
          end
        else
          write(#7) /           {ход неправильный: дать звуковой сигнал }
        end
      else
        correctly := true      {случай EXIT или CHANGE}

```

```

until correctly;
if change then
  GetChange
end; {GetPlayerMove}
{-----}
Procedure SetOwnerMove;
{ Найти и отобразить очередной ход программы }
{-----}
Function CheckField : Integer;
{ Проверка состояния игры. Возвращает 0, если нет ни одной фишки (победа игрока), 1 - есть один ряд (победа машины) и количество непустых рядов в остальных случаях }
var
  i,j : Integer;
begin {CheckField}
  j := 0;
  for i := 1 to nrow do if col[i]>0 then inc(j);
  CheckField := j
end; {CheckField}
{-----}
Procedure CheckPlay;
{ Контроль окончания игры }
var
  i : Integer;
begin {CheckPlay}
  GotoXY(1,25);
  write('Введите 1, если хотите сыграть еще раз, 0 - выход: ');
  readln(i);
  if i=1 then change := true else exit := true
end; {CheckPlay}
{-----}
Procedure PlayerVictory;
{ Поздравить игрока с победой и усложнить игру }
const
  t1 = 'ПОЗДРАВЛЯЮ С ОТЛИЧНОЙ ПОБЕДОЙ!';
var i : Integer;
begin
  GotoXY((80-Length(t1)) div 2,24);
  writeln(t1,#7);
  for i := 1 to nrow do
    if ncol[i]<MAXROW then inc(ncol[i]);
  CheckPlay
end; {PlayerVictory}
{-----}
Procedure OwnVictory;
{ Победа машины }
const
  t1 = 'ВЫ ПРОИГРАЛИ: СЛЕДУЮЩИМ ХОДОМ Я ВЕРУ ВСЕ РЯД ';
var
  i : Integer;
begin {OwnVictory}
  i := 1;
  while col[i]=0 do inc(i);
  GotoXY((80-Length(t1)) div 2,24);
  write(t1,i,#7);
  delay(2000); {задержка на 2 секунды}
  col[i] := 0;
  ShowField;
  CheckPlay

```

```

end; {OwnVictory}
{-----}
Procedure ChooseMove;
{ Выбор очередного хода }
const
  BIT = 6;                                {количество двоичных разрядов}
type
  BitType = array [1..BIT] of Integer;
var
  ncbit : array [1..MAXROW] of BitType;
  i,j,k : Integer;
  nbit : BitType;
{-----}
Procedure BitForm(n : Integer; var b : BitType);
{ Формирует двоичное представление b целого числа n }
var
  i : Integer;
begin {BitForm}
  for i := BIT downto 1 do
    begin
      if odd(n) then b[i] := 1 else b[i] := 0;
      n := n shr 1
    end
  end; {BitForm}
{-----}
begin {ChooseMove}
{Найти двоичное представление количества фишек во всех рядах;}
for i := 1 to nrow do BitForm(col[i],ncbit[i]);
{Найти сумму разрядов по модулю 2;}
for i := 1 to BIT do
  begin
    nbit[i] := 0;
    for j := 1 to nrow do nbit[i] := nbit[i] xor ncbit[j,i]
  end;
{Найти i = старший ненулевой разряд суммы}
  i := 1;
  while nbit[i]=0 do inc(i);
  if i>BIT then
    {Опасный вариант}
    begin
      j := 1;
      while col[j]=0 do inc(j);                {найти ненулевой ряд}
      k := 1                                    {взять из него 1 фишку}
    end
  else
    {Безопасный вариант}
    begin
      j := 1;
      while ncbit[j,i]=0 do inc(j);            {найти нужный ряд}
      for i := i to BIT do
        if nbit[i]=1 then
          ncbit[j,i] := ord(ncbit[j,i]=0);    {инверсия разрядов}
        k := 0;
        for i := 1 to BIT do
          begin
            if ncbit[j,i]=1 then inc(k);
            if i<BIT then k := k shl 1
          end;
        k := col[j] - k
    end;

```

```

end;
GotoXY(1,23);
write('Мой ход:      ');
GotoXY(WhereX-8,WhereY);
delay(1000);
write(j, ' ',k);
col[j] := col[j]-k
end; {ChooseMove}
{-----}
begin {SetOwnerMove}
  case CheckField of
    0 : PlayerVictory;      {проверить количество непустых рядов}
    1 : OwnVictory;         {все ряды пусты - победа игрока}
    else
      ChooseMove;           {один непустой ряд - победа машины}
  end; {case}
end; {SetOwnerMove}
{-----}
begin {Главная программа}
  nrow := 3;                { Подготовить игру }
  ncol[1] := 3;              { на поле из трех }
  ncol[2] := 4;              { рядов фишек }
  ncol[3] := 5;
  repeat                    { Цикл изменения условий игры }
    Prepare;                { Подготовить экран }
    repeat                  { Игровой цикл }
      GetPlayerMove;        { Получить ход пользователя }
      if not (exit or change) then
        SetOwnerMove        { Определить собственный ход }
      until exit or change
    until exit
  end.
end.

```

П5.4. ПРОГРАММА NOTEBOOK

Описание программы см. п.15.

Program Notebook;

{Программа обслуживает файлы данных "записной книжки".}

Описание программы см. в гл.15}

Uses App, Objects, Menus, Drivers, Views, StdDlg, DOS, Memory, Dialogs;

type

{Объект TWorkWin создает рамочное окно с полосами скроллинга для управления встроенным в него объектом TInterior}

PWorkWin = ^TWorkWin;

TWorkWin = object (TWindow)

Constructor Init(Bounds: TRect);

end;

{Объект TDlgWin создает диалоговое окно для выбора режима работы}

PDlgWin = ^TDlgWin;

TDlgWin = object (TDialog)

Procedure HandleEvent(var Event: TEvent); Virtual;

end;

{Следующий объект обслуживает внутреннюю часть рамочного окна TWorkWin. Он создает скроллируемое окно с записями из архивного файла и с помощью диалогового окна TDlgWin управляет работой с этими записями}

PInterior = ^TInterior;

```

TInterior = object (TScroller)
  PS: PStringCollection;
  Location: Word;
  Constructor Init(var Bounds: TRect; HS,VS: PScrollBar);
  Procedure Draw; Virtual;
  Procedure ReadFile;
  Destructor Done; Virtual;
  Procedure HandleEvent(var Event: TEvent); Virtual;
end;

```

{Объект-программа TNotebook поддерживает работу с меню и строкой статуса}

```

TNotebook = object (TApplication)
  Procedure InitStatusLine; Virtual;
  Procedure InitMenuBar; Virtual;
  Procedure HandleEvent(var Event: TEvent); Virtual;
  Procedure FileSave;
  Procedure Changedir;
  Procedure DOSCall;
  Procedure FileOpen;
  Procedure Work;
end;

```

const

```

{Команды для обработчиков событий:}
cmChDir   = 202; {Сменить каталог}
cmWork    = 203; {Обработать данные}
cmDOS     = 204; {Временно выйти в ДОС}
cmCan     = 205; {Команда завершения работы}
cmDelete  = 206; {Уничтожить текущую запись}
cmSearch  = 207; {Искать нужную запись}
cmEdit    = 208; {Редактировать запись}
cmAdd     = 209; {Добавить запись}
{Множество временно недоступных команд:}
WinCom1: TCommandSet = [cmSave,cmWork];
WinCom2: TCommandSet = [cmOpen];

```

```

LName = 25;           {Длина поля Name}
LPhone = 11;          {Длина поля Phone}
LAddr = 40;           {Длина поля Addr}
LLine = LName+LPhone+LAddr; {Длина строки}

```

type

```

DataType = record      {Тип данных в файле}
  Name : String [LName]; {Имя}
  Phone: String [LPhone]; {Телефон}
  Addr : String [LAddr]  {Адрес}
end;

```

var

```

DataFile: file of DataType; {Файловая переменная}
OpFileF : Boolean;           {Флаг открытого файла}

```

```

{-----
Реализация объекта TWorkWin
-----}

```

```

Constructor TWorkWin.Init(Bounds: TRect);
{Создание окна данных}

```

```

var
  HS,VS: PScrollBar;      {Полосы-указатели}
  Interior: PInterior;     {Указатель на управляемое текстовое окно}
begin

```

```

TWindow.Init(Bounds,'',0); {Создаем новое окно с рамкой}
GetClipRect(Bounds);      {Получаем в BOUNDS координаты
                           минимальной перерисовываемой части окна}
Bounds.Grow(-1,-1);       {Устанавливаем размеры окна с текстом}
{Включаем стандартные по размеру и положению полосы-указатели;}
VS := StandardScrollBar(sbVertical+sbHandleKeyboard);
HS := StandardScrollBar(sbHorizontal+sbHandleKeyboard);
{Создаем текстовое окно;}
Interior := New(PInterior,Init(Bounds, HS, VS));
Insert(Interior)          {Включаем его в основное окно}
end; {TWorkWin.Init}
{-----}
Procedure TDlgWin.HandleEvent;
begin
    Inherited HandleEvent(Event);
    if Event.What=evCommand then
        EndModal(Event.Command)
end;
{-----}
Procedure TNotebook.FileOpen;
{Открывает файл данных}
var
    PF: PFileDialog;      {Диалоговое окно выбора файла}
    Control: Word;
    s: PathStr;
begin
    {Создаем экземпляр динамического объекта;}
    New(PF, Init('*.dat','Выберите нужный файл:','Имя файла',fdOpenButton,0));
    {С помощью следующего оператора окно выводится на экран и результат работы
    пользователя с ним помещается в переменную Control;}
    Control := DeskTop^.ExecView(PF);
    {Анализируем результат запроса;}
    case Control of
        StdDlg.cmFileOpen,cmOk:
            begin
                {Пользователь указал имя файла;}
                PF^.GetFileName(s); {s содержит имя файла}
                Assign(DataFile,s);
                {$I-}
                Reset(DataFile);
                if IOResult <> 0 then
                    Rewrite(DataFile);
                OpFileF := IOResult=0;
                {$I+}
                if OpFileF then
                    begin
                        DisableCommands(WinCom2);
                        EnableCommands(WinCom1);
                        Work {Переходим к работе}
                    end
                end;
            end;
        end; {case Control}
        Dispose(PF, Done) {Уничтожаем экземпляр}
    end; {FileOpen}
    {-----}
Procedure TNotebook.FileSave;
{Закрывает файл данных}
begin
    Close(DataFile);
    OpFileF := False;
    EnableCommands(WinCom2); {Разрешаем открыть файл}

```

```

    DisableCommands (WinCom1)           {Запрещаем работу и сохранение}
end; {TNotebook.FileSave}
{-----}
Procedure TNotebook.ChangeDir;
{Изменяет текущий каталог}
var
    PD: PChDirDialog;                   {Диалоговое окно смены каталога/диска}
    Control: Word;
begin
    New(PD, Init(cdNormal,0));           {Создаем диалоговое окно}
    Control := DeskTop^.ExecView(PD);   {Используем окно}
    ChDir(PD^.DirInput^.Data^);         {Устанавливаем новый каталог}
    Dispose(PD, Done)                   {Удаляем окно из кучи}
end; {TNotebook.ChangeDir}
{-----}
Procedure TNotebook.DOSCall;
{Временный выход в ДОС}
const
    txt = 'Для возврата введите EXIT в ответ'+
          ' на приглашение ДОС...';
begin
    DoneEvents;                         {Закреть обработчик событий}
    DoneVideo;                          {Закреть монитор экрана}
    DoneMemory;                         {Закреть монитор памяти}
    SetMemTop(HeapPtr);                 {Освободить кучу}
    WriteLn(txt);                       {Сообщить о выходе}
    SwapVectors;                       {Установить стандартные векторы}
    {Передать управление командному процессору ДОС:}
    Exec(GetEnv('COMSPEC'),'');
    {Вернуться из ДОС:}
    SwapVectors;                       {Восстановить векторы}
    SetMemTop(HeapEnd);                 {Восстановить кучу}
    InitMemory;                        {Открыть монитор памяти}
    InitVideo;                         {Открыть монитор экрана}
    InitEvents;                        {Открыть обработчик событий}
    InitSysError;                      {Открыть обработчик ошибок}
    Redraw                             {Восстановить вид экрана}
end; {DOSCall}
{-----}
Constructor TInterior.Init;
{Создает окно скроллера}
begin
    TScroller.Init(Bounds, Hs, VS);
    ReadFile;
    GrowMode := gfGrowHiX+gfGrowHiY;
    SetLimit(LLine, PS^.Count)
end;
{-----}
Destructor TInterior.Done;
begin
    Dispose(PS, Done);
    Inherited Done
end;
{-----}
Procedure TInterior.ReadFile;
{Читает содержимое файла данных в массив LINES}
var
    k: Integer;
    s: String;
    Data: DataType;

```

```

f: text;
begin
  PS := New(PStringCollection, Init(100,10));
  seek(DataFile,0);
  while not (EOF(DataFile) or LowMemory) do
    begin
      Read(DataFile, data);
      with data do
        begin
          s := Name;
          while Length(s) < LName do
            s := s+' ';
          s := s+Phone;
          while Length(s) < LName+LPhone do
            s := s+' ';
          s := s+Addr
        end;
        if s<>' ' then PS^.Insert(NewStr(S))
      end;
      Location := 0;
    end; {ReadFile}
  {-----}
  Procedure TInterior.Draw;
  {Выводит данные в окно просмотра}
  var
    n,                                {Текущая строка экрана}
    k: Integer;                       {Текущая строка массива}
    B: TDrawBuffer;
    Color: Byte;
    p: PString;
  begin
    if Delta.Y>Location then
      Location := Delta.Y;
    if Location>Delta.Y+pred(Size.Y) then
      Location := Delta.Y+pred(Size.Y);
    for n := 0 to pred(Size.Y) do
      {Size.Y - количество строк окна}
      begin
        k := Delta.Y+n;
        if k=Location then
          Color := GetColor(2)
        else
          Color := GetColor(1);
        MoveChar(B, ' ', Color, Size.X);
        if k < pred(PS^.Count) then
          begin
            p := PS^.At(k);
            MoveStr(B, Copy(p^,Delta.X+1,Size.X),Color);
          end;
        WriteLine(0,N,Size.X,1,B)
      end
    end; {TInterior.Draw}
  {-----}
  Function Control: Word;
  {Получает команду из основного диалогового окна}
  const
    X = 1;
    L = 12;
    DX= 13;

```



```

But: array [0..4] of String [13] =           {Надписи на кнопках;}
('1~ Выход ', '2~ Убрать ', '3~ Искать ', '4~ Изменить ', '5~ Добавить ');
Txt: array [0..3] of String [52] = (
{Справочный текст:}
'Убрать - удалить запись, выделенную цветом',
'Искать - искать запись, начинающуюся нужными буквами',
'Изменить - изменить поле (поля) выделенной записи',
'Добавить - добавить новую запись');

var
  R: TRect;
  D: PDlgWin;
  k: Integer;
begin
  R.Assign(7,6,74,15);
  D := New(PDlgWin, Init(R, 'Выберите продолжение:'));
  with D^ do
    begin
      for k := 0 to 3 do           {Вставляем поясняющий текст}
        begin
          R.Assign(1,1+k,65,2+k);
          Insert(New(PStaticText, Init(R, #3+Txt[k])))
        end;
      for k := 0 to 4 do         {Вставляем кнопки:}
        begin
          R.Assign(X+k*DX,6,X+k*DX+L,8);
          Insert(New(PButton,
            Init(R, But[k], cmCan+k, bfNormal)))
        end;
      SelectNext(False);         {Активизируем первую кнопку}
    end;
    Control := DeskTop^.ExecView(D); {Выполняем диалог}
  end; {Control}
  {-----}
  Procedure TInterior.HandleEvent;
  Procedure DeleteItem;
  {Удаляет указанный в Location элемент данных}
  var
    D: Integer;
    PStr: PString;
    s: String;
    Data: DataType;
  begin
    PStr := PS^.At(Location);      {Получаем текущую запись}
    s := copy(PStr^,1,LName);
    seek(DataFile,0);
    D := -1;                       {D - номер записи в файле}
    repeat                         {Цикл поиска по совпадению поля Name:}
      inc(D);
      read(DataFile,Data);
      with Data do while Length(Name) < LName do
        Name := Name+' ';
      until Data.Name=s;
      seek(DataFile,pred(FileSize(DataFile)));
      read(DataFile,Data);          {Читаем последнюю запись}
      seek(DataFile,D);
      write(DataFile,Data);         {Помещаем ее на место удаляемсй}
      seek(DataFile,pred(FileSize(DataFile)));
      truncate(DataFile);           {Удаляем последнюю запись}
    with PS^ do D := IndexOf(At(Location));
  end;

```

```

    PS^.AtFree(D);
    Draw
end; {DeleteItem}
{-----}
Procedure AddItem(Edit: Boolean);
{Добавляет новый или редактирует старый элемент данных}
const
    y = 1;
    dy = 2;
    L = LName+LPhone+LAddr;
var
    Data: DataType;
    R: TRect;
    InWin: PDialog;
    BName, BPhone, BAddr: PInputLine;
    Control: Word;
    OldCount: Word;
    s: String;
    p: PString;
begin
    Seek(DataFile, FileSize(DataFile));
    repeat
        if Edit then {Готовим заголовок}
            s := 'Редактирование:'
        else
            begin
                Str(FileSize(DataFile)+1, s);
                while Length(s) < 3 do
                    s := '0'+s;
                s := 'Вводится запись N '+s
            end;
        FillChar(Data, SizeOf(Data), ' ');
        R.Assign(15, 5, 65, 16);
        InWin := New(PDialog, Init(R, s));
        with InWin^ do
            begin
                R.Assign(2, y+1, 2+LName, y+2);
                BName := New(PInputLine, Init(R, LName));
                Insert(BName);
                R.Assign(2, y, 2+LName, y+1);
                Insert(New(PLabel,
                    Init(R, 'Имя', BName)));
                R.Assign(2, y+dy+1, 2+LPhone, y+dy+2);
                BPhone := New(PInputLine,
                    Init(R, LPhone));
                Insert(BPhone);
                R.Assign(2, y+dy, 2+LPhone, y+dy+1);
                Insert(New(PLabel, Init(R, 'Телефон', BPhone)));
                R.Assign(2, y+2*dy+1, 2+LAddr, y+2*dy+2);
                BAddr := New(PInputLine, Init(R, LAddr));
                Insert(BAddr);
                R.Assign(2, y+2*dy, 2+LAddr, y+2*dy+1);
                Insert(New(PLabel,
                    Init(R, 'Адрес', BAddr)));
                {Вставляем две командные кнопки:}
                R.Assign(2, y+3*dy+1, 12, y+3*dy+3);
                Insert(New(PButton,
                    Init(R, 'Ввести', cmOK, bfDefault)));
                R.Assign(2+20, y+3*dy+1, 12+20, y+3*dy+3);
                Insert(New(PButton,

```

```

    Init(R, 'Выход', cmCancel, bfNormal));
    SelectNext(False)           {Активизируем первую кнопку}
end;                             {Конец формирования окна}
if Edit then with Data do
begin                             {Готовим начальный текст;}
    p := PS^.At(Location);       {Читаем данные из записи}
    s := p^;
    Name := copy(s, 1, LName);
    Phone := copy(s, succ(LName), LPhone);
    Addr := copy(s, succ(LName+LPhone), LAddr);
    InWin^.SetData(Data)        {Вставляем текст в поля ввода}
end;
Control := DeskTop^.ExecView(InWin); {Выполняем диалог}
if Control=cmOk then with Data do
begin
    if Edit then
        DeleteItem;             {Удаляем старую запись}
        Name := BName^.Data^;
        Phone := BPhone^.Data^;
        Addr := BAddr^.Data^;
        s[0] := chr(L);
        FillChar(s[1], L, ' ');
        move(Name[1], s[1], Length(Name));
        move(Phone[1], s[succ(LName)], Length(Phone));
        move(Addr[1], s[succ(LName+LPhone)], Length(Addr));
        OldCount := PS^.Count;   {Прежнее количество записей}
        PS^.Insert(NewStr(s));   {Добавляем в коллекцию}
        {Проверяем добавление}
        if OldCount <> PS^.Count then
            Write(DataFile, Data) {Да - добавляем в файл}
        end
    until Edit or (Control=cmCancel);
    Draw
end; {AddItem}
{-----}
Procedure SearchItem;
{Ищет нужный элемент}
Function UpString(s: String): String;
{Преобразует строку в верхний регистр}
var
    k: Integer;
begin
    for k := 1 to Length(s) do
        if s[k] in ['a'..'z'] then
            s[k] := chr(ord('A')+ord(s[k])-ord('a'))
        else if s[k] in ['a'..'п'] then
            s[k] := chr(ord('A')+ord(s[k])-ord('a'))
        else if s[k] in ['p'..'я'] then
            s[k] := chr(ord('P')+ord(s[k])-ord('p'));
    UpString := s
end; {UpString}
var
    InWin: PDialog;
    R: TRect;
    s: String;
    p: PInputLine;
    k: Word;
begin {SearchItem}
    R.Assign(15, 8, 65, 16);

```

```

InWin := New(PDialog,
Init(R, 'Поиск записи:'));
with InWin^ do
begin
  R.Assign(2,2,47,3);
  p := New(PInputLine, Init(R,50));
  Insert(p);
  R.Assign(1,1,40,2);
  Insert(New(PLabel, Init(R, 'Введите образец для поиска:',p)));
  R.Assign(10,5,20,7);
  Insert(New(PButton, Init(R, 'Ввести', cmOk, bfDefault)));
  R.Assign(25,5,35,7);
  Insert(New(PButton, Init(R, 'Выход', cmCancel, bfNormal)));
  SelectNext(False)
end;
if DeskTop^.ExecView(InWin) = cmCancel then
  exit;
s := p^.Data^;
Location := 0;
while (UpString(s) >= UpString(PString(PS^.At(Location))^))
  and (Location < pred(PS^.Count)) do
  inc(Location);
if (Location < Delta.Y) or (Location > Delta.Y+pred(Size.Y)) then
  ScrollTo(Delta.X, Location)
else
  Draw
end; {SearchItem}
{-----}
var
  R: TPoint;
label Cls;
begin
  TScroller.HandleEvent(Event);
  case Event.What of
  evCommand:
    case Event.Command of
    cmClose:
      begin
Cls:
        case Control of {Получить команду из основного диалогового окна}
        cmCan,
        cmCancel: EndModal(cmCancel);
        cmEdit : AddItem(True);
        cmDelete: DeleteItem;
        cmSearch: SearchItem;
        cmAdd : AddItem(False);
        end
      end;
        cmZoom: exit;
      end;
    evMouseDown: {Реакция на щелчок мышью}
      begin
        MakeLocal(MouseWhere, R); {Получаем в R локальные
                                   координаты указателя мыши}

        Location := Delta.Y+R.Y;
        Draw
      end;
    evKeyDown: {Реакция на клавиши + -}
      case Event.KeyCode of

```

```

kbEsc: goto Cls;
kbGrayMinus: if Location > Delta.Y then
  begin
    dec(Location);
    Draw
  end;
kbGrayPlus: if Location < Delta.Y+pred(Size.Y) then
  begin
    inc(Location);
    Draw
  end;
end
end
end; {TInterior.HandleEvent}
{-----}
Procedure TNotebook.Work;
{Работа с данными}
var
  R: TRect;
  PW: PWorkWin;
  Control: Word;
begin
  R.Assign(0,0,80,23);
  PW := New(PWorkWin, Init(R));
  Control := DeskTop^.ExecView(PW);
  Dispose(PW,Done)
end;
{-----}
Procedure TNotebook.HandleEvent(var Event: TEvent);
{Обработчик событий программы}
begin {TNotebook.HandleEvent}
  TApplication.HandleEvent(Event); {Обработка стандартных
                                     команд cmQuit и cmMenu}

  if Event.What = evCommand then
  case Event.Command of
    {Обработка новых команд:}
    cmOpen: FileOpen; {Открыть файл}
    cmSave: FileSave;  {Закрыть файл}
    cmChangeDir : ChangeDir; {Сменить диск}
    cmDOSShell : DOSCall; {Временный выход в ДОС}
    cmWork : Work; {Обработать данные}
  else
    exit {Не обрабатывать другие команды}
  end;
  ClearEvent(Event) {Очистить событие после обработки}
end; {TNotebook.HandleEvent}
{-----}
Procedure TNotebook.InitMenuBar;
{Создание верхнего меню}
var
  R: TRect;
begin
  GetExtent(R);
  R.B.Y := succ(R.A.Y); {R - координаты строки меню}
  MenuBar := New(PMenuBar, Init(R,
    NewMenu( {Создаем меню}
      {Первый элемент нового меню представляет собой
      подменю (меню второго уровня). Создаем его}
      NewSubMenu('~F~/Файл', hcNoContext,
        {Описываем элемент главного меню}

```

```

NewMenu(
    NewItem(
        '~1~/ Открыть', 'F3', kbF3, cmOpen, hcNoContext,
        NewItem(
            '~2~/ Закрыть', 'F2', kbF2, cmSave, hcNoContext,
            NewItem(
                '~3~/ Сменить диск', '', 0, cmChangeDir, hcNoContext,
                NewLine(
                    '~4~/ Вызов ДОС', '', 0, cmDOSShell, hcNoContext,
                    NewItem('~5~/ Конец работы', 'Alt-X', kbAltX, cmQuit, hcNoContext,
NIL))))))
),
{Создаем второй элемент главного меню}
NewItem('~W~/ Работа', '', kbF4, cmWork, hcNoContext,
NIL) {Нет других элементов главного меню}
)))
end; {TNotebook.InitMenuBar}
{-----}
Procedure TNotebook.InitStatusLine;
{Формирует строку статуса}
var
    R: TRect; {Границы строки статуса}
begin
    GetExtent(R); {Получаем в R координаты всего экрана}
    R.A.Y := pred(R.B.Y);
    StatusLine := New(PStatusLine,
    Init(R,
        NewStatusDef(0, $FFFF,
        NewStatusKey('~Alt-X~ Выход', kbAltX, cmQuit,
        NewStatusKey('~F2~ Закрыть', kbF2, cmSaveFile,
        NewStatusKey('~F3~ Открыть', kbF3, cmOpenFile,
        NewStatusKey('~F4~ Работа', kbF4, cmWork,
        NewStatusKey('~F10~ Меню', kbF10, cmMenu,
        NIL)))))
    ));
    DisableCommands(WinCom1)
end; {TNotebook.InitStatusLine}
{-----}
var
    Nbook: TNotebook;
begin
    Nbook.Init;
    Nbook.Run;
    Nbook.Done
end.

```

Литература

1. Абель П. Язык Ассемблера для IBM PC и программирования/Пер. с англ. Ю.В.Сальникова. — М.: Высш. шк., 1992. — 447 с., ил.
2. Аваков С.М. Системные функции DOS 4.0./Библиотека информационной технологии: Вып. 4./Под ред. Г.Р.Громова. — М.: ИнфоАрт, 1992. — С. 3...98.
3. Белецкий Я. Турбо Паскаль с графикой для персональных компьютеров/Пер. с польск. Д.И.Юренкова. — М.: Машиностроение, 1991. — 320 с.
4. Береза Д.А. Графические шрифты фирмы Borland. «Мир ПК», N8, 1992. — С. 109...114.
5. Блашкин И.И., Буров А.А. Новые возможности Turbo—Pascal 6.0. — СПб.: Изд-во «Макет», 1992. — 64 с.
6. Борзенко А. Путешествие по памяти. «Компьютер Пресс», N7, 1992. — С. 7...14.
7. Бородин Ю.С. и др. Паскаль для персональных компьютеров: Справ. пособие/Ю.С.Бородин, А.Н.Вальвачев, А.И.Кузьмич. — Мн.: Высш. шк.: БФ ГИТМП «НИКА», 1991. — 365 с.
9. Васильев П.П. Турбо Паскаль — мой друг: М.: Компьютер, ЮНИТИ, 1995. — 96 с.
9. Данкан Р. Профессиональная работа в MS—DOS: Пер. с англ. — М.: Мир, 1993. — 509 с., ил.
10. Джордейн Р. Справочник программиста персональных компьютеров типа IBM PC, XT и AT: Пер. с англ./Предисл. Н.В.Гайского. — М.: Финансы и статистика, 1991. — 544 с.
11. Зуев Е.А. Язык программирования Turbo Pascal 6.0. — М.: Унитех, 1992. — 298 с.
12. Касперский Е. Компьютерные вирусы в MS—DOS. — М.: «Эдель», 1992. — 176 с.
13. Мизрохи. Turbo Pascal и объектно—ориентированное программирование. — М.: Финансы и статистика, 1992. — 185 с.
14. Моисеенков И. Суэта вокруг Роберта или Моррис—сын и все, все, все... — «Компьютер Пресс», N8, 1991. — С. 45...62
15. Мурашко И.В., Авалян В.Э. Библия MS—DOS версии 5.0 в 2—х книгах. Кн. 1. — М.: НПО «Гермес», 1992. — 220 с.
16. Мурашко И.В., Авалян В.Э. Библия MS—DOS версии 5.0 в 2—х книгах. Кн. 2. — М.: НПО «Гермес», 1992. — 251 с.
17. Основский Ю.Н. Компьютерные вирусы. Классификация, методы борьбы. — М.: Центр МИФИ СП Диалог, 1990. — 41 с.
18. Правиков Д.И. Можно ли защититься от нелегального копирования?./Библиотека информационной технологии: Вып. 4./Под ред. Г.Р.Громова. — М.: ИнфоАрт, 1992. — С. 99...107.
19. Расторгуев С.П., Дмитриевский Н.Н. Искусство защиты и «раздевания» программ. — М.: Совмаркет, 1991. — 94 с.
20. Скэнлон Л. Персональные ЭВМ IBM PC и XT. Программирование на языке ассемблера: Пер. с англ. — 2—е изд., стереотип. — М.: Радио и связь, 1991. — 336 с., ил.
21. Спесивцев А.В. и др. Защита информации в персональных ЭВМ. — М.: Радио и связь, МП «Веста», 1992. — 192 с.
22. Справочник по процедурам и функциям Borland Pascal with Objects 7.0. — Киев: «Диалектика», 1993. — 272 с.
23. Фаронов В.В. Программирование на персональных ЭВМ в среде Турбо—Паскаль. — М.: Изд-во МГТУ, 1990. — 580 с.
24. Фаронов В.В. Турбо Паскаль (в 3—х книгах). Кн.1. Основы Турбо Паскаля. — М.: Учебно—инженерный центр «МВТУ—ФЕСТО ДИДАКТИК», 1992. — 304 с.
25. Фаронов В.В. Турбо Паскаль (в 3—х книгах). Кн.2. Библиотека Turbo Vision. — М.: Учебно—инженерный центр «МВТУ—ФЕСТО ДИДАКТИК», 1993. — 412 с.
26. Фаронов В.В. Турбо Паскаль (в 3—х книгах). Кн.3. Практика программирования. Часть 1. — М.: Учебно—инженерный центр «МВТУ—ФЕСТО ДИДАКТИК», 1993. — 256 с.
27. Фаронов В.В. Турбо Паскаль (в 3—х книгах). Кн.3. Практика программирования. Часть 2. — М.: Учебно—инженерный центр «МВТУ—ФЕСТО ДИДАКТИК», 1993. — 304 с.
28. Фаронов В.В. Практика Windows—программирования. — М.: Информпечать, 1996. — 247 с.
29. Фаронов В.В. Паскаль и Windows. — М.: Учебно—инженерный центр «МВТУ—ФЕСТО ДИДАКТИК», 1995. — 539 с.
30. Фролов А.В., Фролов Г.В. Аппаратное обеспечение IBM PC: В 2—х ч. Ч. 1. — М.: «ДИАЛОГ—МИФИ», 1992. — 208 с.

31. Фролов А.В., Фролов Г.В. Аппаратное обеспечение IBM PC: В 2-х ч. Ч. 2. — М.: «ДИАЛОГ-МИФИ», 1992. — 208 с.
32. Фролов А.В., Фролов Г.В. Операционная система MS-DOS: В 3-х кн. Кн. 3. — М.: «ДИАЛОГ-МИФИ», 1991. — 224 с.
33. Фролов А.В., Фролов Г.В. Защищенный режим процессоров Intel 80286, 80386, 80486. Практическое руководство по использованию защищенного режима. — М.: «ДИАЛОГ-МИФИ», 1993. — 240 с. (Библиотека системного программиста, т.6).
34. Фролов А.В., Фролов Г.В. MS-DOS для программиста. Часть 2. — М.: «ДИАЛОГ-МИФИ», 1995. — 256 с. (Библиотека системного программиста, т.19).
35. Федоров А. Особенности программирования на Borland Pascal. — Киев: Диалектика, 1994. — 144 с.
36. Хершель Р. Турбо Паскаль/2-е изд., перераб. — Вологда: МП «МИК», 1991. — 342 с.
37. Хижняк. Пишем вирус и... антивирус./Под общей редакцией И.М.Овсянниковой. — М.: ИНТО, 1991. — 90 с.
38. Шерстюк Ф.Н. Вирусы и антивирусы на компьютере IBM PC./ Библиотека информационной технологии: Вып.2. /Под ред. Г.Р.Громова. — М.: ИнфоАрт, 1991. — С. 119...138.
39. POWER TOOLS PLUS. Процедуры поддержки для Turbo Pascal 4.0: Справочное руководство пользователя. Техническая документация. — 318 с.

*Готовится
к выпуску!*

П.В. Шумаков, В.В. Фаронов **Технология разработки приложений баз данных на Delphi 4**



Эта книга рассматривает один из важнейших аспектов современного программирования - работу с базами данных. Книга разбита на две части. В первой рассматриваются общие вопросы построения приложений и средства, не зависящие

от выбранной архитектуры базы данных. Во второй части книги рассматриваются вопросы построения приложений в архитектуре «клиент-сервер».

Изложение ведется на основе исследования возможностей SQL-сервера Borland InterBase (локальная версия которого поставляется вместе с Delphi). Параллельно показывается, как те или иные средства серверной базы данных могут использоваться в приложениях Delphi.

В.А. Шамис

Borland C++ Builder 3.

Программирование без проблем

Книга посвящена
наиболее
перспективному
компилятору C++.
В ней рассмотрены
среда данного
компилятора и его
особенности,
современные
методы
программирования,
реализованные для
C++, стройная
компонентная



модель, расширенные средства управления
проектами, технология двусторонней интеграции
приложений, синхронизация средств визуального
и текстового редактирования, а также удобный
встроенный отладчик - благодаря чему
C++Builder 3 предоставляет собой
впечатляющую среду разработки. Книга
снабжена большим количеством оригинальных
примеров и будет полезна широкому кругу
программистов.

В.П. Дьяконов и И.В. Абраменкова

MathCAD 7.0

в математике, физике и в Internet



Книга посвящена применению новейшей системы символьной математики MathCAD 7.0 PRO в массовых математических расчетах. При этом в качестве примеров применения системы взяты расчеты, встречающиеся при решении типовых задач в курсе физики для вузов. Впервые описана практика работы с системой в глобальной

сети Internet. Такой подход позволяет избавиться от абстрактности вычислений, принятой в справочной литературе по математическим расчетам и способствует развитию у учащихся интереса к математическим аспектам решения физических и научно-технических задач. Книга может использоваться в качестве краткого справочного пособия по решению задач математической физики с применением современных компьютерных математических систем. Для студентов и преподавателей вузов и университетов, а также для всех пользователей ПК, заинтересованных в их серьезном применении.



Д.А. Рамодин
Borland JBuilder.

***Программирование на Java
без проблем***



Отличительной особенностью содержащегося в книге материала является наличие фрагментов работающего проверенного кода, иллюстрирующего рассказы. Из этой книги пользователь узнает о том, как работать со средой разработчика JBuilder, как пользоваться ее

мастерами, как создавать пользовательский интерфейс с применением менеджеров расположения (layouts), характерных только для языка программирования Java. Читатель почерпнет знания о том, как создавать системы управления базами данных, и в заключение научится создавать свои собственные компоненты JavaBeans.

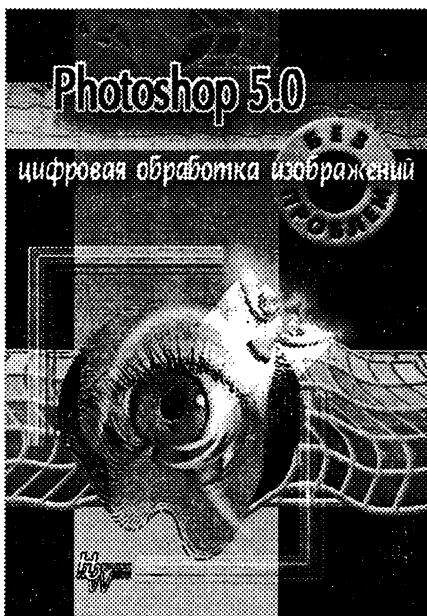
Иван Деруго *Photoshop 5.0.*

*Готовится
к выпуску!*

Цифровая обработка изображений без проблем

Не будет преувеличением сказать, что программный пакет Photoshop фирмы Adobe является в настоящее время общепризнанным стандартом для цифровой обработки изображений. А в руках опытного пользователя, будь он фотохудожником, журналистом или веб-дизайнером, этот продукт вообще превращается в идеальное средство

для редактирования любых электронных изображений, последующей их обработки (сюда смело можно включить практически весь “багаж” современного фотографа, привыкшего добиваться дополнительных эффектов с помощью химических веществ и механических процессов), восстановления поврежденных изображений, ретуширования фотографий, составления самых фантастических коллажей, которые только может позволить себе наше воображение.



«ДЛЯ ЗАМЕТОК»

«ДЛЯ ЗАМЕТОК»

«ДЛЯ ЗАМЕТОК»

с. 278 - Уреженик,

Ваше имя: _____

Возраст: ☐ до 16 ☐ 16-25 ☐ 25-40 ☐ после 40

Образование: ☐ среднее ☐ незак. высшее ☐ высшее

В настоящее время Вы:

- ☐ студент ☐ ваша основная работа связана с компьютерами
☐ работаете, Ваша работа не связана с компьютерами
☐ безработный

Конфигурации Вашего компьютера:

Процессор (производитель)

☐ Intel ☐ x86 (AMD, Ciryx, IDC, IBM) ☐ PowerPC ☐ не знаю

Процессор (мощность)

☐ 386 и ниже ☐ 486 ☐ Pentium ☐ Pentium MMX ☐ Pentium II
☐ Celeron ☐ другое _____

RAM

☐ до 8 Мб ☐ 12-16 Мб ☐ 20-32 Мб ☐ 40-64 Мб ☐ больше 64 Мб

Используемая для работы среда

(отметьте все используемые):

☐ DOS ☐ Win3.1 ☐ Win95 ☐ Win98 ☐ WinNT
☐ UNIX ☐ OS/2 ☐ LINUX ☐ Mac OS ☐ другое _____

Имеется ли компьютер

☐ дома ☐ только на работе ☐ и дома, и на работе

Имеете доступ к Internet, вид связи:

☐ модемная ☐ выделенная ☐ нет

Ваш провайдер _____

Имеется ли у Вас собственная страничка ☐ да ☐ нет ☐ создается

Ее адрес _____

Опыт работы:

☐ нет опыта ☐ начинающий пользователь
☐ опытный пользователь ☐ программист
☐ администратор ☐ тех. специалист

Область интересов:

☐ office и бухгалтерия ☐ графика и анимация
☐ верстка ☐ игры
☐ информационные ☐ программирование
 технологии и Internet ☐ web-design
☐ работа с БД ☐ сетевые технологии
☐ аппаратное обеспечение
☐ другое _____

Какие версии программ Вы предпочитаете использовать?

☐ оригинальные (англоязычные) ☐ локализованные (русифицированные)
☐ Без разницы

Расставьте, пожалуйста, в приоритетном порядке (1–7) значимость для Вас источника получения информации по работе на компьютер:

- | | |
|--|------------------------------------|
| <u>4</u> интернет, электронные конференции | <u>3</u> книги |
| <u>4</u> компьютерные журналы | <u>1</u> курсы, выставки, семинары |
| <u>6</u> встроенные описания (help) | <u>2</u> коллеги, друзья, знакомые |
| <u>5</u> мультимедиа-продукты (tutorial) | |

Какие программные средства, выпущенные в последнее время, кажутся вам наиболее интересными и перспективными?

Продуктами каких фирм Вы интересуетесь?

- | | | | | |
|---|---|---|--------------------------------|----------------------------------|
| ☒ Microsoft | ☒ Adobe | ☒ Corel | ☐ Quark | ☐ Borland |
| ☐ Symantec | ☐ Netscape | ☐ Autodesk | ☐ Lotus | ☐ Novell |
| ☐ Macromedia | ☐ Oracle | ☐ Отечественных | | |
| ☐ другое _____ | | | | |

Какие интересные и необходимые в работе книги вы приобрели бы, если бы увидели в продаже? Какой софт недостаточно освещен в печати? _____

Если Вы начинаете пользоваться новым продуктом, всегда ли покупаете книги по этому продукту?

- ☐ никогда ☐ иногда ☐ всегда
☐ если не могу разобраться ☒ если есть деньги

Запоминаете ли Вы название издательства, книги которых покупаете?

- ☒ да ☐ нет

Какое издательство компьютерной литературы, на Ваш взгляд, издает наибольшее количество интересных книг?

Покупали ли Вы другие книги нашего издательства? ☒ да ☐ нет

Как Вы считаете, более интересны книги:

- ☒ отечественных авторов ☐ переводные

Обращаете ли Вы на это внимание при покупке? ☒ да ☐ нет

Могли бы Вы стать автором или соавтором книги?

- ☐ да ☐ да при наличии свободного времени ☒ нет

Тема Вашей возможной работы _____

Если возможно, укажите, пожалуйста, наиболее интересные сайты компьютерных новостей, сайты с полезными программами _____

Какой из компьютерных журналов наиболее полно освещает круг интересующих Вас проблем? _____

Благодарим Вас!

Валерий Васильевич Фаронов
Турбо Паскаль 7.0. Начальный курс.
Учебное пособие

Редактор Шамис В. А.

ЛР № 064480 от 4 марта 1996 г. Подписано в печать 02.03.99.
Формат 70 × 100/16. Печать высокая. Бумага газетная.
Усл. печ. л. 39. Тираж 15 000 экз. Зак. № 1962.

Издательство «Нолидж»
Москва, Новоданиловская наб., д. 9
тел.: 235-05-07

Отпечатано с готовых диапозитивов в ГМП
«Первая Образцовая типография»
Государственного комитета РФ по печати.
113054, Москва, Валовая, 28.

Эту, а также более 3-х тысяч других книг по компьютерным технологиям, экономике, юриспруденции и психологии вы можете приобрести в Москве по следующим адресам:

- Фирма "Ридас", М. Тульская, Новоданиловская наб., 9, тел. 954-30-44, 742-59-86
- Филиал фирмы "Ридас", М. Текстильщики, Саратовская ул. 8/1, 4-й подъезд, тел. 919-62-10.

Или заказать по минимальным ценам у наших региональных представителей:

- г.Владивосток «Эста» (4232) 29-43-20, 51-76-53, 33-07-45 факс
- г.Волгоград «Техническая книга» (8442) 36-35-97
- г.Воронеж УБК «Барокко» (0732) 78-93-37
- г.Донецк «Дом книги» 0622-55-7449
- г.Иваново Магазин ООО «Мысль» (0932) 32-60-81
- г.Иркутск «Продалит» (3952) 27-10-67
- г.Н.Новгород ООО «Дефис» 8-8312-37-24-64
- г.Новосибирск «Деловая литература» (3832) 35-48-42
- г.Омск ООО ТД «Собрание сочинений» (3812) 401635, 652409, 333-9000
- г.Самара «Чакона» (8462) 36-14-87, 41-39-57, 41-02-21
- г.Санкт-Петербург «Символ плюс» 812-311-0626
- г.Тамбов Магазин ТОО «Богема» 8-075-2-47-27-72
- г.Тюмень Магазин «Знание-сила»
- г.Уфа Книжный магазин «Букинист №15» (3472) 22-63-04
- г.Хабаровск «Мирс» (4212) 34-30-29, 22-73-30 факс
- г.Челябинск «Вива» (3512) 34-57-67

С нашим ассортиментом Вы также сможете ознакомиться на Internet-сервере:

WWW.READUS.RU

РИДАС • READ US

Издательство и книжная торговля

представляет широкий ассортимент специальной литературы по:

- | | |
|--|----------------------------|
| 📖 персональным компьютерам и программированию (>800 наим.) | 📖 бухучету (> 500 наим.) |
| 📖 электронике и технике (>300 наим.) | 📖 праву (>500 наим.) |
| 📖 экономике (>500 наим.) | 📖 учебники (>100 наим.) |
| 📖 психологии (>500 наим.) | 📖 справочники (>300 наим.) |

Различные формы и сроки оплаты, гибкая система скидок, в том числе и для мелкооптовых покупателей.

У нас только ИЗДАТЕЛЬСКИЕ ЦЕНЫ.

Подробнее с нашим ассортиментом и условиями сотрудничества можно ознакомиться на сервере
WWW.READUS.RU.

Приглашаем к сотрудничеству авторов и издательства, региональных представителей, рекламодателей.

Наши телефоны:

тел./факс: (095) 954-3044, тел.: 742-5986

Адрес: Москва, Варшавское шоссе, д. 9

Вход со стороны набережной.



www.readus.ru

Условия сотрудничества фирмы "Ридас" (вступают в действие с 01.09.97)

1) Для всех категорий клиентов оплачивающих отгруженную продукцию в течении 3-х дней, устанавливаются скидки в зависимости от объема заказа в размере:

а)

Сумма	Ассортимент	Книги со спец. скидкой (в прайсе помечены*)
от 200 до 1000 руб.	1%	1%
от 1000 руб.	3%	3%
от 2000 руб.	5%	5%
от 3000 руб.	7%	9%
от 5000		

Телефон: (095) 954-3044, (095) 952-8839
Адрес: Варшавское шоссе дом 9

Готовится к выпуску!
Об этой книге.
Прежде чем приступить к написанию этого текста, автор попытался представить своего будущего читателя. А так как круг пользователей, хоть раз в жизни запускавших на своем компьютере программу с названием Adobe Photoshop бесконечно широк (с уверенностью можно заключить, что слышали о подобной программе абсолютно все), то в результате пришлось мысленно соединиться с обобщенным образом человека, который для своей работы с цифровыми изображениями выбирает самое лучшее. Ориентируясь на то, что "опыт" — сын ошибок трудных" и пути познания каждого индивидуума несподручны, автор попытался создать некую квинтэссенцию сухого справочника и пересказа многого опыта. Отметить те отправные точки, опираясь на которые, любознательный читатель захочет сам экспериментировать с программой. Автор уверен, что только в творческом союзе разработчиков и пользователей может родиться по-настоящему ценный и функциональный продукт. На примере некоторых опытов автор постарается показать, что порой даже программист, непосредственно осуществивший в программе ту или иную возможность, не знал, что он на самом деле создал более совершенный инструмент.

www.knowledge.ru

www.books.ru

TurboPascal 7.0

ISBN 5-89251-054-9



9 785892 510547 >